

Conference Proceedings



*UNIX is a Registered Trademark of AT&T

San Francisco, CA
June 20 - 24 1988

Summer
1988

San Francisco,
California

CONFERENCE
PROCEEDINGS

USENIX Association

**Proceedings of the
Summer 1988 USENIX Conference**

**June 20-24, 1988
San Francisco, California USA**

For additional copies of these proceedings, write:

USENIX Association

P.O. Box 2299

Berkeley, CA 94701 USA

Price: \$20.00 plus \$25.00 for overseas mail

Copyright 1988 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain
with the author or the author's employer.

UNIX is a registered trademark of AT&T.

Other trademarks are noted in the text.

TABLE OF CONTENTS

Preface	vii
Conference Committee	viii

PLENARY SESSION

Wednesday (9:00-10:30)

Chair: Peter Salus

KEYNOTE ADDRESS: The Object is the Standard
Adele Goldberg, ParcPlace Systems

WINDOW SYSTEMS

Wednesday (11:00-12:30)

Chair: Robert Scheifler

Using the X Toolkit or How to Write a Widget	1
<i>Joel McCormack & Paul Asente, Digital Equipment Corporation</i>	
G Shell Environment	15
<i>Rick Macklem, Jim Linders, Hugh Smith, University of Guelph</i>	
X11/NeWS Design Overview	23
<i>Robin Schaufler, Sun Microsystems, Inc.</i>	

FILE SYSTEMS

Wednesday (2:00-3:30)

Chair: Jim McKie

Pseudo Devices: User-Level Extensions to the Sprite File System	37
<i>Brent B. Welch & John K. Ousterhout, University of California, Berkeley</i>	
A UNIX File System for a Write-Once Optical Disk	51
<i>Terry Laskodi, Bob Eifrig, Jason Gait, Tektronix, Inc.</i>	
The File Motel – An Incremental Backup System for UNIX	61
<i>Andrew Hume, AT&T Bell Laboratories</i>	

TOOLS

Wednesday (4:00-5:30)

Chair: David Tilbrook

SPIFF – A Program for Making Controlled Approximate Comparisons of Files	73
<i>Daniel Nachbar, Bell Communications Research</i>	
Rtools: Tools for Software Management in a Distributed Computing Environment	85
<i>Helen E. Harrison, Stephen P. Schaefer, Terry S. Yoo, Microelectronics Center of North Carolina</i>	
Tools and Policies for the Hierarchical Management of Source Code Development	95
<i>Thomas Lord, Carnegie Mellon University</i>	

SECURITY I

Thursday (9:00-10:30)

Chair: Alex Morrow

Retaining SUID Programs in a Secure UNIX	107
<i>Steven M. Kramer, SecureWare, Inc.</i>	
Extending the UNIX Protection Model with Access Control Lists	119
<i>Gary Fernandez & Larry Allen, Apollo Computer, Inc.</i>	
Experience Adding C2 Security Features to UNIX	133
<i>Matthew S. Hecht, Abhai Johri, Radhakrishna Aditham, T. John Wei, IBM Systems Integration Division</i>	

PROGRAMMING LANGUAGES

Thursday (11:00-12:30)

Chair: Ken Arnold

γ-GLA: A Generator for Lexical Analyzers That Programmers Can Use	147
<i>Robert W. Gray, University of Colorado, Boulder</i>	
Saber-C: An Interpreter-Based Programming Environment for the C Language	161
<i>Stephen Kaufer, Russell Lopez, Sessa Pratap, Saber Software, Inc.</i>	
Associative Arrays in C++	173
<i>Andrew Koenig, AT&T Bell Laboratories</i>	

SYSTEMS I

Thursday (2:00-3:30)

Chair: Rob Gingell

Copy-on-Write for Sprite	187
<i>Michael Nelson & John Ousterhout, University of California, Berkeley</i>	
A Strategy for SMP ULTRIX	203
<i>Ursula Sinkewicz, Digital Equipment Corporation</i>	
A Heap-Based Callout Implementation to Meet Real-Time Needs	213
<i>Ronald E. Barkley, AT&T Information Systems; T. Paul Lee, AT&T Bell Laboratories</i>	

RUNTIME ENVIRONMENTS

Thursday (4:00-5:30)

Chair: Jay Lepreau

A Memory Allocation Profiler for C and Lisp Programs	223
<i>Benjamin Zorn & Paul Hilfinger, University of California, Berkeley</i>	
A RISC Approach to Runtime Exceptions	239
<i>Mark Himmelstein, MIPS Computer Systems, Inc.; Steven Correll, Key Computer Laboratories; Kevin Enderby, NeXT Inc.</i>	
CASPER the Friendly Daemon	251
<i>Ronald E. Barkley, AT&T Information Systems; Danny Chen, AT&T Bell Laboratories</i>	

SECURITY II

Friday (9:00-10:30)

Chair: Dave Presotto

NIDX – A Real-Time Intrusion Detection Expert System	261
<i>David S. Bauer & Michael E. Koblenz, Bell Communications Research</i>	
Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses	275
<i>Nick Lai, University of California, Los Angeles; Terence E. Gray, 3Com Corporation</i>	
Concurrent Access Licensing	287
<i>S. Margaret Olson, Paul H. Levine, Stuart H. Jones, Stephanie Bodoff, Stephen C. Bertrand, Apollo Computer, Inc.</i>	

SYSTEMS II

Friday (11:00-12:30)

Chair: Miche Baker-Harvey

Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel	295
<i>Marshall Kirk McKusick & Michael J. Karels, University of California, Berkeley</i>	
A Dynamic UNIX Operating System	305
<i>Robert Rodriguez, Matt Koehler, Larry Palmer, Ricky Palmer, Digital Equipment Corporation</i>	
Stellix: UNIX for a Graphics Supercomputer	321
<i>Thomas J. Teixeira, Robert F. Gurwitz, Stellar Computer, Inc.</i>	

TEXT AND TERMINALS

Friday (2:00-3:30)

Chair: Dave Lennert

Prototext: Universal Text Drivers	331
<i>V. Guruprasad, Kirloskar Computer Services Ltd.</i>	
The "Session Tty" Manager	339
<i>S.M. Bellovin, AT&T Bell Laboratories</i>	
A Low Cost Bitmapped Terminal on the Atari ST	355
<i>Peter Collinson, University of Kent, Canterbury</i>	

NETWORKING

Friday (4:00-5:30)

Chair: John Mashey

GATED: A Multi-Routing Protocol Daemon for UNIX	365
<i>Mark S. Fedor, NYSERNet Inc.</i>	
Asmodeus: A Daemon Servant for the System Administrator	377
<i>Mark E. Epstein, Curt Vandetta, John Sechrest, Oregon State University</i>	
Big Brother: A Network Services Expert	393
<i>Don Peacock & Mark Giuffrida, University of Michigan</i>	

PREFACE

Welcome to the 1988 Summer USENIX Technical Conference and Exhibition. These proceedings contain the 33 papers presented in the Technical Program. Unlike previous USENIX conferences, the papers for this conference were selected through a peer review process that judged *full papers* on their technical merit and potential interest to the conference attendees. (Previous conferences used only abstracts.) We (the program committee) are pleased with the results of this selection, including the paper judged as the best student submission; "A Memory Allocation Profiler for C and Lisp Programs", by Benjamin Zorn of the University of California, Berkeley. Since the process by which submissions were judged and selected was very different for this conference, let me describe it for you.

Organizing the review process actually began prior to the 1987 Summer conference in Phoenix. Since this was the first time that full papers would be judged for a USENIX conference, I felt it was important that we look at similar procedures used by other organizations. In particular, I contacted Maureen Stone, the technical program chair for the ACM SIGGRAPH '87 conference. Maureen and her assistant Subhana Menis were kind enough to provide us with the instructions and review forms, they used in organizing their conference, and these were then converted to a form suitable for our use.

The call for papers went out about the time of the Phoenix Conference, June 1987, and the deadline for submissions was February 19th of this year. Nearly all the papers arrived within 2 days of the deadline. Most papers arrived by courier; one paper was hand-delivered by the author who bicycled across the Golden Gate Bridge to Marin!

For the next few days I sifted through the submissions reading as much as was necessary to categorize each paper according to its subject matter. Papers were then assigned to the members of the Technical Program Committee, who act as senior reviewers for the papers. We do our best to match papers with interested reviewers.

Each senior reviewer then received a complete copy of all papers submitted, 3 copies of their assigned papers, review instructions, and scoring forms. They were then responsible for redistributing the papers to at least two other qualified reviewers that represent at least two other unrelated views. Each paper was also reviewed by another member of the program committee.

Two weeks before the Technical Program Committee met to discuss the papers each senior reviewer was responsible for collecting their reviews. These reviews were then processed by the senior reviewers into a set of scores which were transmitted to the program chair. Most of these came in on time. With all the scores in hand, I was then able to rank all the papers. This ranking was returned to each committee member so that they could prepare themselves for the Technical Program Committee meeting by reading those papers that were "on the edge".

The program committee met in Berkeley this year (for the first time ever at the USENIX headquarters). We typically spend two days selecting papers and organizing the technical program, but this year were able to complete our work in only one day. During the meeting we discussed the relative merits of papers, listened to reviews, and made fun of how nervous the program chair was. By the end of our meetings we had selected the best papers and organized the agenda. This process was extremely draining and I thank all the members of the program committee for their work.

Once papers had been selected, my job began in earnest. All authors were notified of the results of the selection process. Each author received a copy of the review forms for their papers as well as any pages from their submission(s) that were marked by the reviewers. Suggestions generated at the program committee meeting were also returned. Authors of accepted papers were expected to revise their paper according to the reviewers' comments and return a final copy to the Proceedings Organizer. This year Evi Nemeth was responsible for the proceedings, riding herd over the authors (requesting second versions when authors did not follow the 'do what I mean, not what I say' principle in interpreting the authors instructions), generating all the front matter, and, in general, doing all the necessary odds and ends that make the proceedings happen. I think Evi did a wonderful job, thank you.

I must also thank Deirdre Warin of Pixar who helped me during many of the tasks I described above. Finally, thanks are due to Pixar for allowing me the time to perform my duties as program chair.

Sam Leffler
Technical Program Chair

CONFERENCE COMMITTEE

CONFERENCE ORGANIZERS

Samuel J. Leffler, *Technical Program*
(Pixar)
Judith F. Desharnais, *Meeting Planner*,
(USENIX Association)
Evi Nemeth, *Proceedings Production*
(University of Colorado, Boulder)
John Donnelly, *Tutorial Coordinator*
(USENIX Association)
John Donnelly, *Vendor Exhibition Manager*
(USENIX Association)

TECHNICAL PROGRAM COMMITTEE

Ken Arnold (Apollo Computers)
Miche Baker-Harvey (Digital Equipment Corp.)
Steven J. Buroff (AT&T)
Robert A. Gingell (Sun Microsystems)
Samuel J. Leffler, *Chair* (Pixar)
Dave Lennert (Hewlett-Packard)
Jay Lepreau (University of Utah)
Jim McKie (Bellcore)
John Mashey (MIPS Computer Systems, Inc.)
Barton Miller (University of Wisconsin)
Alex Morrow (IBM Academic Inf. Systems)
David L. Presotto (AT&T Bell Laboratories)
Robert W. Scheifler (MIT Laboratory for CS)
David Tilbrook (Information Tech. Center, CMU)

TECHNICAL PROGRAM REVIEWERS

Jaap Akkerhuis	Ray Lanza
Eric Allman	Jim Lawson
Miche Baker-Harvey	Bob Lenk
J. M. Barton	Dan Levin
Steven Bellovin	Tom Libert
Beth Benoit	Jeff Lind
Mark J. Bradakis	Mark Linton
Carl Braganza	Suzanne Logcher
Lawrence M. Breed	Eric Lund
Phyllis Bregman	Paul C. Lustgarten
Greg Buzzard	Chris Maio
Brian Byun	Eva Manolis
Robert Chesler	Joe Martin
Jong-Deok Choi	Teri McConnell
David Cohrs	Kirk McKusick
Clement T. Cole	Rae McLellan
Don Cragun	Lee McMahon

Gregory Depp
Jeffrey C. Doughty
William J. Earl
David C. Elliott
Peter Ford
R Don Freeman
Jim Fulton
Steve Gaede
Anat Gafni
Jim Gettys
David Goldberg
Ron Gomes
Alan Graham
Fred Grampp
Gerry Green
Daniel E. Greer, Jr.
Jeffrey A. Gumpf
Harley Hahn
Graham Hamilton
Gordon Harris
Guy Harris
Carol Hatcher
Tim Hayes
Mike Hibler
Kee Hinckley
David Hitz
Brian Holt
Peter Honeyman
Tom Houghton
Nat Howard
Irene Hu
Ed Hunter
Mike Kazar
Jerry Keselman
Douglas P. Kingston
Dave Kristol
John LaPorta
Peter Langston

Bill Melohn
Doug Merritt
Paul Mielke
Joseph P. Moran
Evi Nemeth
Marco Papa
Mike Paquette
Flip Phillips
Michael L. Powell
Mike Powell
John Quartman
Peggy Quinn
Steve Rago
Daniel L. Reading
Steve Reber
Brian Redman
Jon Reeves
Dennis Ritchie
Jon Rosenberg
Mike Russell
Mike Scheer
Tony Schene
Debbie Scherrer
Michael Schueller
Jim Seagraves
Donn Seeley
Barbara Shapiro
Phil Sherin
Bob Sidebotham
Keith Sklower
Leigh Stoller
Ralph R. Swick
Michael Thompson
Dan Tiernan
Howard Trickey
J. C. Wagner
Larry Weber
Yung Weiliu

PROCEEDINGS PRODUCTION

Evi Nemeth, Dotty Foerst, Trent Hein,
and Paul Kooros (University of Colorado)

AUTHOR INDEX

Radhakrishna Aditham	133	Andrew Koenig	173
Larry Allen	119	Steven M. Kramer	107
Paul Asente	1	Nick Lai	275
Ronald E. Barkley	213	Terry Laskodi	51
Ronald E. Barkley	251	T. Paul Lee	213
David S. Bauer	261	Paul H. Levine	287
S. M. Bellovin	339	Jim Linders	15
Stephen C. Bertrand	287	Russell Lopez	161
Stephanie Bodoff	287	Thomas Lord	95
Danny Chen	251	Rick Macklem	15
Peter Collinson	355	Joel McCormack	1
Stephen Correll	239	Marshall Kirk McKusick	295
Mark E. Epstein	377	Daniel Nachbar	73
Bob Eifrig	51	Michael Nelson	187
Kevin Enderby	239	S. Margaret Olson	287
Mark S. Fedor	365	John Ousterhout	187
Gary Fernandez	119	John K. Ousterhout	37
Jason Gait	51	Larry Palmer	305
Mark Giuffrida	393	Ricky Palmer	305
Robert W. Gray	147	Don Peacock	393
Terence E. Gray	275	Sesha Pratap	161
V. Guruprasad	331	Robert Rodriguez	305
Robert F. Gurwitz	321	Stephen P. Schaefer	85
Helen E. Harrison	85	Robin Schaufler	23
Matthew S. Hecht	133	John Sechrest	377
Paul Hilfinger	223	Ursula Sinkewicz	203
Mark Himmelstein	239	Hugh Smith	15
Andrew Hume	61	Thomas J. Teixeira	321
Abhai Johri	133	Curt Vandetta	377
Stuart H. Jones	287	Brent B. Welch	37
Michael J. Karels	295	Terry S. Yoo	85
Stephen Kaufer	161	T. John Wei	133
Michael E. Koblenz	261	Benjamin Zorn	223
Matt Koehler	305		

Using the X Toolkit or How to Write a Widget

Joel McCormack

Western Research Laboratory

joel@decwrl.dec.com

Paul Asente

Western Software Laboratory

asente@decwrl.dec.com

Digital Equipment Corporation

Abstract

The X11 Window System defines a network protocol [4] for communication between a graphics server and an application. The X library [1] provides a procedural interface to the protocol, in which most calls are a thin veneer over the underlying protocol operations.

The X toolkit [2] is an object-oriented construction kit built on top of the X library. The toolkit is used to write user interface components ("widgets"), to organize a set of widget instances into a complete user interface, and to link a user interface with the functionality provided by an application.

This paper briefly describes the goals and structure of the X toolkit, shows a small application program that uses the toolkit, then shows two of the widgets used by the application.

1. An Introduction to X

The X11 Window System defines a distributed, asynchronous protocol [4] by which graphics servers and applications communicate. A graphics server can support multiple applications, and an application can use multiple servers.

An application and a server can run on the same or on different machines. The X protocol assumes a server and an application are connected by a fast communication link like shared memory, Ethernet, or even a leased line. To minimize the effects of network latency, the X protocol is asynchronous: neither the server nor the application wait for acknowledgements. The X library occasionally feigns synchronicity for the convenience of the application; for example, the application can query the mouse position or ask for a new colormap entry by calling a procedure that waits for a reply from the server.

The X library [1] provides a procedural interface to the protocol. An application calls procedures in the X library to send window management and drawing commands to the server. The server sends event notifications to the application in response to user actions (moving the mouse, typing on the keyboard) and screen geometry changes (window exposure, size or position change, iconification). The X library queues events and packages them into a record structure; an application periodically polls the library for the next event.

2. Why a Toolkit?

The X library provides a powerful low-level interface, but this flexibility has a cost: even simple programs are hard to write. The primary goal of the X toolkit [2] is to reduce the effort needed to write an X application. While a program to write "Hello, world" based directly on the X library takes 40 executable statements [3], the equivalent toolkit program takes 5. The toolkit enables programmers to write a library of widgets like text labels, scroll bars, command buttons, and menus, and to assemble these widgets into a complete user interface with just a few lines of code.

A secondary goal is to allow user customization of applications. Users can specify resources like colors, fonts, border widths, and sizes for any widget or for various sets of widgets with just a few lines of text in a user preferences file.

Finally, when an application synchronously queries the server it suffers the latency inherent in a round trip—hundreds of these queries quickly add up to several seconds. The toolkit extensively caches data on the application side to minimize this time.

3. Intrinsic, Widgets, and Applications

The *intrinsic* layer of the toolkit is a mostly policy-free foundation upon which widgets and applications are built. The intrinsic contains facilities to create, organize, and destroy widgets; to translate event sequences from the server into procedure calls that applications and widgets have registered; to read and write state maintained by a widget; and to negotiate over screen real estate when a widget changes size or position. It also includes a few predefined widgets that deal with much of the boiler-plate data and code common to all widgets.

A *widget* is a user interface component implemented using calls to the intrinsic and the X library. MIT's Project Athena delivers a sample set of widgets [5] with the toolkit intrinsic, which serve as examples of how to write commonly seen user interface components. The sample widgets can be replaced, customized, or used as is. Many applications use only existing widgets; a few supplement these with their own specialized widgets.

An *application* uses the intrinsic to tie widgets together into a user interface and to bind this user interface to functions implemented by the application. Applications typically interact with X only through the toolkit. An application programmer doesn't embed specialized user interface components in an application, but instead writes new widgets that can be reused and shared with other programmers.

4. How an Application Writer Sees the World

An application consists of three conceptual parts: application functionality, user interface, and links between the two. This division makes it easier to modify the user interface without recoding the functionality. In actual code, the division between the interface and its linkage to the functionality is not always clearly visible: the statements that link the user interface to the application's function are often intertwined with the construction of that interface.

The application functionality is a set of "callback" routines the toolkit invokes in response to user actions. The xmh mail handler distributed by Project Athena has routines to include new mail, read a message, reply to a message, and so forth. When a user clicks on a command button in the user interface, the command button widget calls one or more of these routines.

The user interface is a tree of widget instances. Figure 4-1 shows the tree created by the example program in figure 5-2. The root and internal nodes are composite widgets. A composite widget usually doesn't have any display or input semantics: it just manages the size and position of its children. The leaves of the tree are primitive widgets. A primitive widget cannot have children: it just displays portions of its state and responds to user input.

An application links its functionality to its user interface by binding callbacks and data structures to widget instances. When the intrinsics invoke a callback routine, they pass the widget that caused the invocation, data registered with the toolkit when the application bound the callback, and widget-specific data. The application uses the application data if it needs to associate additional information with the widget. The format of the widget data is defined by the particular widget. A scroll bar's "slider-changed" callback passes the address of a structure containing the new slider position, while a command button callback supplies no extra information and thus passes NULL.

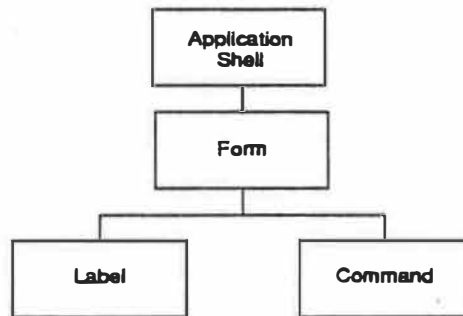


Figure 4-1: The widget instance tree for the "Goodbye, world" program.

5. A Simple Application: Goodbye, world.

"Hello, world" has earned itself a beloved niche in the hearts of all C programmers. Times change, however, so to show the capabilities of the toolkit we present a more sophisticated example: the "Goodbye, world" program. It displays a window with a label and a command button; when the command button is clicked with the mouse the program prints a farewell message and terminates (Figure 5-1).

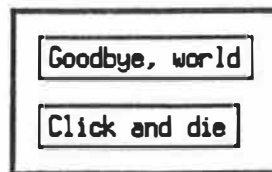


Figure 5-1: The output of the "Goodbye, world" program

Our example shows the main program and simple but complete versions of the **Label** and **Command** widgets it uses. Our description of the code omits some details due to space limitations, and instead concentrates on the general flavor of toolkit programming. Refer to the intrinsics documentation [2] for further information.

```

#include "Intrinsic.h"
#include "StringDefs.h"
#include "Form.h"
#include "Label.h"
#include "Command.h"

void Quit(widget, clientData, callData)
    Widget widget;
    caddr_t clientData, callData;
{
    (void) printf("Goodbye, cruel world\n");
    exit(0);
}

int main(argc, argv)
    unsigned int argc;
    char **argv;
{
    Widget shell, form, label, command;
    Arg arg[25];
    unsigned int n;

    shell = XtInitialize("goodbye", "Goodbye", NULL, 0, &argc, argv);

    form = XtCreateManagedWidget("form", formWidgetClass, shell, (Arg *) NULL, 0);

    n = 0;
    XtSetArg(arg[n], XtNx, 10);          n++;
    XtSetArg(arg[n], XtNy, 10);          n++;
    XtSetArg(arg[n], XtNlabel, "Goodbye, world"); n++;
    label = XtCreateManagedWidget("label", labelWidgetClass, form, arg, n);

    n = 0;
    XtSetArg(arg[n], XtNx, 10);          n++;
    XtSetArg(arg[n], XtNy, 40);          n++;
    XtSetArg(arg[n], XtNlabel, "Click and die"); n++;
    command = XtCreateManagedWidget("command", commandWidgetClass, form, arg, n);
    XtAddCallback(command, XtNcallback, Quit, NULL);

    XtRealizeWidget(shell);
    XtMainLoop();
}

```

Figure 5-2: Goodbye.c: The "Goodbye, world" program.

Figure 5-2 is the goodbye application. "Intrinsic.h" contains the intrinsic definitions needed by applications, "StringDefs.h" contains predefined strings used as resource names, and "Form.h", "Label.h", and "Command.h" contain the definitions needed to use these widgets.

The callback procedure `Quit` needs no additional information, so it ignores its parameters. It prints an exit message and terminates the program.

The main program declares four widgets, *shell*, *form*, *label*, and *command*. Each application needs a special top level widget called a *Shell* that holds exactly one child and communicates with the window manager. Our application has two children, so it puts them in the widget *form* and puts *form* in *shell*.

The application first calls the routine `XtInitialize`. The first two parameters are the name and class of the application, which `XtInitialize` uses to read in user preferences for application and widget resources. These preferences can be overridden by the command line arguments passed in as the last two parameters. `XtInitialize` establishes a connection with the server and returns a *Shell* widget for the application to use.

Main creates the container widget, a **Form** widget named *form*, as a child of *shell*¹. **XtCreateManagedWidget** takes five parameters: the name of the widget instance, the class of the widget, a widget instance to use as the parent, a list of arguments to the widget, and the length of this list. *Form* has no extra arguments and so passes **NULL** and 0 as the last two parameters.

To create the **Label** widget, the application provides values for the label's *x*, *y* and *label* resources by passing a list of name-value pairs. **XtSetArg** is used to create this list in a stylized coding metaphor that makes it easier to add, delete, or change the settings in the code without making mistakes. This specification mechanism may seem cumbersome, but it has the great advantage that only the resources that the application cares about need to be specified. The intrinsics require widgets to provide default values for all resources; for complex widgets with dozens of resources the programming convenience is significant. **XtCreateManagedWidget** creates a new widget, called *label*, using *form* as its parent and the assigned resources to override default values.

The application creates the **Command** widget similarly, then attaches the callback procedure **Quit** to it using **XtAddCallback**. When the command button is activated, the toolkit calls **Quit** and the program terminates.

6. How a Widget Writer Sees the World

Sometimes an application writer needs to write a new widget. Often it will be a slight variation on an existing widget, but it can be something completely different. In either case, the toolkit allows the writer to ignore features the new widget has in common with an existing widget and concentrate on the differences.

To accomplish this, the toolkit supports a single-inheritance class hierarchy. Each widget type is a class with a single superclass and possibly many subclasses. A subclass contains both the declarations of its superclass and data and routines to implement its additional functionality. A subclass can inherit procedures (called "methods") from its superclass or can implement equivalent semantics itself. Any operation that is valid for a class is valid for all subclasses.

Figure 6-1 shows an abbreviated class hierarchy of the widget set distributed by Project Athena. The toolkit intrinsics define the four special classes **Core**, **Composite**, **Constraint**, and **Shell**. All other classes are implemented as direct or indirect subclasses of these.

The top of the class hierarchy is the **Core** widget, which contains declarations and code that are common to all widgets. It declares the parent widget pointer, name, size and position, background, and border color, and defines the methods to maintain this data.

The **Composite** widget is the superclass for widgets that can contain children. It contains methods for adding and deleting children and for negotiating with them about size and placement.

The **Constraint** widget is the superclass for composite widgets that maintain extra constraints on a per-child basis. For example, many constraint subclasses keep minimum and maximum allowable sizes for each child.

¹This example uses the **Form** widget to be distributed by MIT in X11 Version 3. To make it work on Version 2, replace "XtNx" and "XtNy" with "XtNhorizDistance" and "XtNvertDistance".

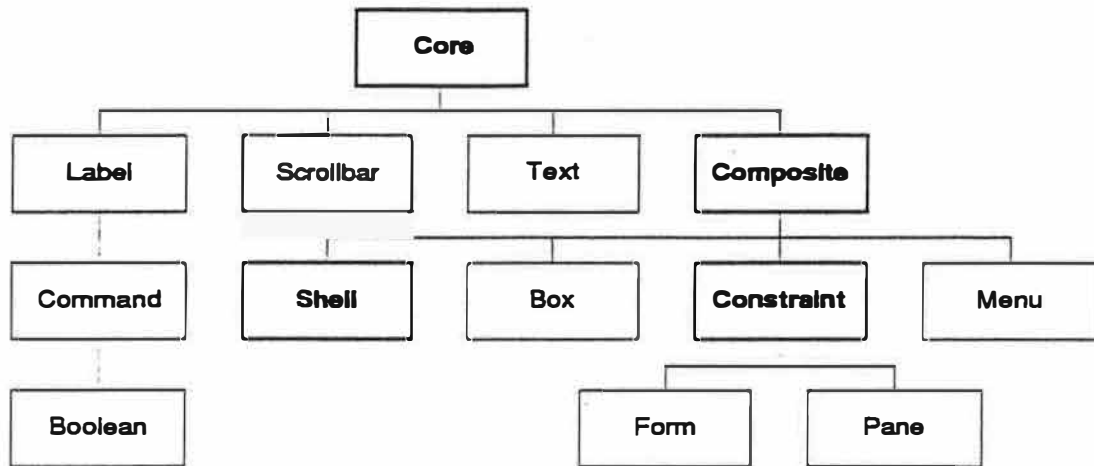


Figure 6-1: The class tree for the Project Athena widget set.

The **Shell** widget provides the interface between the internally consistent, cached world of the toolkit and the external world of X windows. Shell widgets communicate with the window manager and are used to “pop up” other widgets like menus and dialog boxes.

To create a new widget class, a programmer first looks at the existing class hierarchy to determine if something similar already exists. If the new widget is similar to an existing widget he creates a subclass of that widget; otherwise he creates a direct subclass of **Core**, **Composite**, or **Constraint**;

The widgets in the next section show both types of subclass. **Label** is a subclass of **Core** and requires many lines of declarations and code. **Command** is just like **Label**, but adds a pinch of extra semantics. It is a subclass of **Label** and requires relatively little code—most of its functionality is inherited from **Label**.

7. A Simple Label Widget

A widget definition is split into three files: a public header file containing information needed to use the widget, a private header file containing information needed to subclass the widget, and an implementation file with the code that defines the widget’s behavior.

```

#define XtNforeground    "foreground"
#define XtNlabel         "label"
#define XtNfont          "font"

typedef struct _LabelRec *LabelWidget;
extern WidgetClass labelWidgetClass;
  
```

Figure 7-1: Label.h: The label public header file

Figure 7-1 shows the public header file for **Label**. It defines the names of the three resources **Label** adds to **Core** (**XtNforeground**, **XtNlabel**, and **XtNfont**); a type definition for **Label** widget instances; and an external declaration for the class structure, used in **XtCreateManagedWidget**.

```

#include "Label.h"

typedef struct {
    int make_compiler_happy;
} LabelClassPart;

typedef struct _LabelClassRec {
    CoreClassPart    core_class;
    LabelClassPart   label_class;
} LabelClassRec, "LabelWidgetClass";

extern LabelClassRec labelClassRec;

typedef struct {
    Pixel            foreground;
    XFontStruct      *font;
    char             *label;
    GC               normal_GC;
    Dimension        label_width, label_height;
    int              label_len;
} LabelPart;

typedef struct _LabelRec {
    CorePart         core;
    LabelPart        label;
} LabelRec;

#define WIDTH_PAD 4
#define HEIGHT_PAD 2

```

Figure 7-2: LabelP.h: The label private header file

Figure 7-2 is the private header file for **Label** and contains the class and instance data structure declarations. **Label**'s class structure consists of the Core class part, included in all widgets, and **Label**'s own class part. **Label**'s class part needs no extra information, but since the C compiler cannot handle empty structures it has a *make_compiler_happy* field.

The **LabelPart** record has the fields needed for each **Label** instance. The fields *foreground*, *font*, and *label* correspond to the resources defined in the public header file. The other fields are derived by the widget implementation and are described later. All **Label** instances get a complete **LabelRec** consisting of the core fields common to all widgets and the special fields used by **Label**.

Figure 7-3 is the implementation of **Label**. Its header files include "IntrinsicP.h," the intrinsics header file for widget programmers, and "LabelP.h," the label private header file.

The *resources* array contains the definitions for the label-specific resources. Each entry contains, among other things, the resource name (**XtNforeground**), its type (**XtRPixel**), its offset in the widget record (**XtOffset(...)**), its default value ("DefaultForeground"), and the type of its default (**XtRString**). The intrinsics initializes new records with these defaults unless the values are overridden in the argument list passed to the creation routine or in the user preferences file.

The next part is the initializer for **Label**'s class record. The *superclass* field is the address of the class record for **Label**'s superclass. The class initialization procedures perform initialization specific to this class; since **Label** has none these are NULL. The *initialize* procedure is called to initialize **Label** instances. *Realize* gets called to create the widget's window; here it is inherited from its superclass because **Label** doesn't do anything special. The *resources* and *num_resources* fields describe **Label**'s resource list. *Destroy*, *resize*, and *expose* are called when the widget is destroyed, resized, or needs redisplay. The *set_values* procedure gets called

```

#include <stdio.h>
#include <string.h>
#include "IntrinsicP.h"
#include "LabelP.h"
#include "StringDefs.h"

static XtResource resources[] = {
    {XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
     XtOffset(LabelWidget, label.foreground), XtRString, "DefaultForeground"},

    {XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct *),
     XtOffset(LabelWidget, label.font), XtRString, "fixed"},

    {XtNlabel, XtCLabel, XtRString, sizeof(String),
     XtOffset(LabelWidget, label.label), XtRString, NULL},
};

static void Initialize(), Redisplay(), Destroy();
static Boolean SetValues();

LabelClassRec labelClassRec = {
    {
        /* superclass */           (WidgetClass) &widgetClassRec,
        /* class_name */           "Label",
        /* widget_size */          sizeof(LabelRec),
        /* class_initialize */      NULL,
        /* class_part_initialize */ NULL,
        /* class_inited */         FALSE,
        /* initialize */            Initialize,
        /* initialize_hook */       NULL,
        /* realize */               XtInheritRealize,
        /* actions */               NULL,
        /* num_actions */          0,
        /* resources */             resources,
        /* num_resources */        XtNumber(resources),
        /* xrm_class */             NULLQUARK,
        /* input_flags */           TRUE, TRUE, TRUE, FALSE,
        /* destroy */               Destroy,
        /* resize */                XtInheritResize,
        /* expose */                Redisplay,
        /* set_values */            SetValues,
        /* set_values_hook */       NULL,
        /* set_values_almost */     XtInheritSetValuesAlmost,
        /* get_values_hook */       NULL,
        /* accept_focus */          NULL,
        /* version */               XtVersion,
        /* callback offsets */       NULL,
        /* tm_table */              NULL,
        /* query_geometry */        NULL,
    }
};

WidgetClass labelWidgetClass = (WidgetClass) &labelClassRec;

static void SetTextWidthAndHeight(lw)
    LabelWidget lw;
{
    lw->label.label_len = strlen(lw->label.label);
    lw->label.label_height = lw->label.font->max_bounds.ascent + lw->label.font->max_bounds.descent;
    lw->label.label_width = XTextWidth(fs, lw->label.label, lw->label.label_len);
}

```

Figure 7-3: Label.c: The label implementation file, part 1

```

static void GetNormalGC(lw)
    LabelWidget lw;
{
    XGCValues values;

    values.foreground = lw->label.foreground;
    values.font = lw->label.font->fid;
    lw->label.normal_GC = XtGetGC((Widget) lw, GCForeground | GCFont, &values);
}

static void Initialize(request, new)
    Widget request, new;
{
    LabelWidget lw = (LabelWidget) new;

    if (lw->label.label == NULL) lw->label.label = strcpy(XtMalloc(strlen(lw->core.name) + 1), lw->core.name);

    GetNormalGC(lw);
    SetTextWidthAndHeight(lw);

    if (lw->core.width == 0) lw->core.width = lw->label.label_width + 2 * WIDTH_PAD;
    if (lw->core.height == 0) lw->core.height = lw->label.label_height + 2 * HEIGHT_PAD;
}

static void Redisplay(w, event)
    Widget w;
    XEvent *event;
{
    LabelWidget lw = (LabelWidget) w;

    XDrawString(XtDisplay(w), XtWindow(w), lw->label.normal_GC, WIDTH_PAD,
        HEIGHT_PAD + lw->label.font->max_bounds.ascent, lw->label.label, lw->label.label_len);
}

static Boolean SetValues(old, request, new)
    Widget old, request, new;
{
    LabelWidget oldlw = (LabelWidget) old, newlw = (LabelWidget) new;

    if (newlw->label.label == NULL) newlw->label.label = newlw->core.name;

    if ((oldlw->label.label != newlw->label.label || oldlw->label.font != newlw->label.font) SetTextWidthAndHeight(newlw);

    if (oldlw->label.label != newlw->label.label) {
        if (newlw->label.label != NULL) newlw->label.label = strcpy(XtMalloc(newlw->label.label_len+1), newlw->label.label);

        XtFree((char *) oldlw->label.label);

        if (oldlw->core.width == newlw->core.width) newlw->core.width = newlw->label.label_width + 2 * WIDTH_PAD;
        if (oldlw->core.height == newlw->core.height) newlw->core.height = newlw->label.label_height + 2 * HEIGHT_PAD;
    }

    if (oldlw->label.foreground != newlw->label.foreground || oldlw->label.font->fid != newlw->label.font->fid) {
        GetNormalGC(newlw);
        XtDestroyGC(oldlw->label.normal_GC);
    }
    return TRUE;
}

```

Figure 7-3, continued

```
static void Destroy(w)
Widget w;
{
    LabelWidget lw = (LabelWidget) w;

    XtDestroyGC(lw->label.normal_GC);
    XtFree(lw->label.label);
    XFreeFontInfo((char **) NULL, lw->label.font, 0);
}
```

Figure 7-3, concluded

when the application tries to change any of the widget's state.

The two routines **SetTextWidthAndHeight** and **GetNormalGC** are internal routines to maintain the derived fields in the widget record.

When the application creates a new **Label** widget, the intrinsics call **Initialize**. It uses the instance name as the text to be displayed if none was provided. After computing the derived fields, it checks if the application specified the **Label**'s *width* or *height* and if not computes appropriate values based upon the pixel size of the text of the *label* field. The **Initialize** procedure for **Label** need not compute values for the fields declared in *CorePart*—the **Initialize** procedure defined for **Core** has that responsibility. However, **Label** can override any values in *CorePart* as needed.

Redisplay displays the label text by calling **XDrawString**.

SetValues is called when the application changes the contents of **Label**. Its arguments include *old*, the widget as it is now, and *new*, a copy of the widget with the requested changes. **SetValues** must make *new* a consistent widget by updating all derived fields, allocating storage for changed pointer fields, and freeing the storage for old pointer values.

It again uses the instance name for the label text if necessary. If the text or font has changed, **SetValues** recomputes the text dimensions. If the label text has changed, it allocates a copy of the text, frees the old storage, and, unless the user has specifically overridden the size, recomputes the widget size. Finally it creates a new graphics context and destroys the old one if the foreground or font has changed. **SetValues** need not update any fields declared in *CorePart*.

The last routine, **Destroy**, gets called when a **Label** widget is destroyed and frees all the allocated storage associated with it. Once again, **Destroy** does not need to free any storage associated with the *CorePart* fields.

8. A Simple Command Widget

Command is quite similar to **Label**, and so needs only a bit of extra code. Its public and private header files, Figures 8-1 and 8-2 are analogous to those for **Label**.

```
typedef struct _CommandRec *CommandWidget;
extern WidgetClass commandWidgetClass;
```

Figure 8-1: Command.h: The command button public header file

Figure 8-3 is the implementation of the command widget.

Some widget class information is *chained* by the intrinsics, meaning that the intrinsics automatically use information in the superclass to supplement information in the superclass. The resource list is chained data: a subclass has all its superclass's resources in addition to its own.


```

#include "Command.h"

typedef struct _CommandClass {
    int make_compiler_happy;
} CommandClassPart;

typedef struct _CommandClassRec {
    CoreClassPart      core_class;
    LabelClassPart     label_class;
    CommandClassPart   command_class;
} CommandClassRec, *CommandWidgetClass;

extern CommandClassRec commandClassRec;

typedef struct {
    XtCallbackList      callback_list;
} CommandPart;

typedef struct _CommandRec {
    CorePart            core;
    LabelPart           label;
    CommandPart         command;
} CommandRec;

```

Figure 8-2: CommandP.h: The command button private header file

Initialize, *set_values*, and *destroy* are chained procedures: the superclass's corresponding procedure is called in addition to the subclass's. Chained procedures are never explicitly inherited; if a class has nothing to add to its parent it just specifies NULL. The **Command** widget has NULL *initialize* and *set_values* procedures, and adds its own *destroy* procedure to label's.

The intrinsics do not automatically call superclass procedures for *unchained* operations like *realize*, *resize*, and *expose*. A class can inherit its superclass's operation by specifying a procedure like *XtInheritRealize* in its class record. **Command** inherits all unchained operations from **Label**.

Command is different from **Label** in that it has an additional resource, a callback, and a translation. A full paper could be written on translations, so suffice it to say that the *translations* and *actionsList* tables instruct the intrinsics to call **Command**'s **Notify** procedure if the user presses the left mouse button in the widget's window. **Notify** invokes the callback that the application previously associated with the widget, passing NULL as the widget data.

Command's only other procedure is **Destroy**, which frees its dynamic storage.

The implementation file for **Command** effectively demonstrates the power of subclass inheritance. By building upon the declarations and code for the **Core** and **Label** classes, the additional effort to implement the **Command** class is small.

```

#include <stdio.h>
#include "IntrinsicP.h"
#include "LabelP.h"
#include "CommandP.h"
#include "StringDefs.h"

static XtResource resources[] = {
    (XtNcallback, XtCCallback, XtRCallback, sizeof(caddr_t),
     XtOffset(CommandWidget, command.callback_list), XtRCallback, (caddr_t) NULL),
};

static void Notify(), Destroy();

static char translations[] = "<Button1 Down>:      notify()";

static XtActionsRec actionsList[] = { {"notify",      Notify} };

CommandClassRec commandClassRec = {
    {
        /* superclass          */ (WidgetClass) &labelClassRec,
        /* class_name          */ "Command",
        /* widget_size         */ sizeof(CommandRec),
        /* class_initialize     */ NULL,
        /* class_part_initialize */ NULL,
        /* class_inited         */ FALSE,
        /* initialize          */ NULL,
        /* initialize_hook      */ NULL,
        /* realize              */ XtInheritRealize,
        /* actions              */ actionsList,
        /* num_actions          */ XtNumber(actionsList),
        /* resources            */ resources,
        /* num_resources        */ XtNumber(resources),
        /* xrm_class            */ NULLQUARK,
        /* input flags          */ TRUE, TRUE, TRUE, FALSE,
        /* destroy              */ Destroy,
        /* resize               */ XtInheritResize,
        /* expose               */ XtInheritExpose,
        /* set_values           */ NULL,
        /* set_values_hook      */ NULL,
        /* set_values_almost    */ XtInheritSetValuesAlmost,
        /* get_values_hook      */ NULL,
        /* accept_focus         */ NULL,
        /* version              */ XtVersion,
        /* callback offsets     */ NULL,
        /* tm_table             */ translations,
        /* query_geometry       */ NULL,
    }
};

WidgetClass commandWidgetClass = (WidgetClass) &commandClassRec;

static void Notify(w, event)
    Widget w;
    XEvent *event;
{
    XtCallCallbacks(w, XtNcallback, NULL);
}

static void Destroy(w)
    Widget w;
{
    XtRemoveAllCallbacks(w, XtNcallback);
}

```

Figure 8-3: Command.c: The command button implementation file

9. Conclusion

The X protocol and library support a powerful model of an application's interaction with a graphics device, but provide too primitive an interface for even the simplest application to use conveniently.

By trading off some flexibility for a large gain in simplicity, the X toolkit makes writing an application that uses existing widgets easy. The toolkit takes care of low-level bookkeeping, leaving the application writer free to concentrate on application functionality and user interface design.

When an application writer must create a specialized widget, the toolkit's object-oriented class system makes this task easy, too. The widget's superclass takes care of common functionality, leaving the widget writer free to concentrate on what is new and different. Once written, this new widget can be used in other applications.

References

1. Jim Gettys, Ron Newman, Robert W. Scheifler. *Xlib - C Language Interface*. Massachusetts Institute of Technology, 1987.
2. Joel McCormack, Paul Asente, Ralph Swick. *X Toolkit Library - C Language Interface*. Massachusetts Institute of Technology, 1987.
3. David S. H. Rosenthal. A Simple X.11 Client Program, or, How hard can it really be to write 'Hello, World'? Conference Proceedings, Usenix, Winter, 1987, pp. 229.
4. Robert W. Scheifler. *X Window System Protocol*. Massachusetts Institute of Technology, 1987.
5. Ralph R. Swick and Mark S. Ackerman. The X Toolkit: more Bricks for Building User-Interfaces, or, Widgets for Hire. Conference Proceedings, Usenix, Winter, 1987, pp. 221.

G Shell Environment

Rick Macklem, Jim Linders, Hugh Smith
rick@uogvax2.Bitnet, jgl@uogvax2.Bitnet, hugh@uogvax2.Bitnet
University of Guelph

ABSTRACT

Unix[†] running on the current generation of workstations offers support for sophisticated window systems such as the X Window System[‡] from MIT. However, the visual interface made possible by these window systems is at odds with most current Unix utilities, which assume an ASCII terminal interface. Currently, the solution to this problem requires the window system to support a *terminal emulation* window to which the Unix utilities are attached via *pseudo terminals*. The principal intent of the G Shell Environment is to facilitate the migration of standard Unix utility user interaction from a *try interface* to a *visual interface* based on the X Window System.

Of the 250 plus Unix utilities commonly available, only a small fraction perform significant user interaction. The majority perform short tasks such as C compiles or directory listings without user interaction. A well designed visual top level interface (shell) should be able to provide a run time framework sufficient for these utilities. The Xgsh X client described in this paper is designed to fulfill this task. The interactive utilities such as text editors, mail or conferencing systems require significant changes to make effective use of the workstation's visual interactive capabilities. Ideally, they would all be rewritten to use a visual interface on the window sys however the Visual Interface Filter (VIF) is proposed as an interim solution. A key component of both of these utilities is that they are configuration driven, providing a fair degree of flexibility and permitting them to be used as both a prototyping and production tool.

This paper describes the use of the VIF and Xgsh for the realization of a visual interface for use within a Unix student teaching environment.

Introduction

The advent of Unix based workstations with sophisticated window systems such as MIT's X Window System has opened up the potential for a new realm of use interaction within Unix. This however, is juxtaposed with the 250 plus standard Unix utilities currently in use that were written to interact with a standard ASCII terminal. In fact, a vast majority of the utilities don't even know about a screen display and interact in the style reminiscent of the hardcopy teletype. The two programs described in this paper are an effort to build a visual interface environment around standard Unix utilities so that user interaction may be migrated away from the *try mode* of interaction without a need to re-implement these utilities.

Currently, the window systems on Unix workstations provide a *try emulation* window which is connected to a standard utility via a *pseudo terminal*. The resultant interface offers little, if any, functionality beyond that of the ASCII terminal. The first of the two programs described in this paper (the VIF) is an enhanced *try emulator* that permits a variety of mouse inputs and also I/O stream character sequences to be translated into specified actions including certain graphical outputs to provide a limited visual interface.

[†] Unix is a trademark of AT&T Bell Laboratories

[‡] X Window System is a trademark of Massachusetts Institute of Technology

The second program is a visual interface shell (the Xgsh) that is intended to replace the single line command capabilities of a standard Unix shell by providing its functional equivalent. It in turn runs the Unix command line attached via a pseudo terminal connection to an enhanced terminal emulation window provided by the VIF.

A critical aspect of both of these programs is that they are configurable by the system administrator so that they can provide the kind of environment required by a specific user group. This was done in an effort to permit the generation of a simple, and therefore easy interface, without imposing an overly restrictive structure. As such, the onus is on the configuration builder to find a set of appropriate operations such that the interface fulfills the user's needs but remains sufficiently simple to avoid the need for complex interactions. A structured hierarchy of selection panels, and frequent mouse warping to a default selection, is done in an effort to give the interface a logical conversational flow. This structure is somewhat more defined than the *desk top metaphor* commonly used for screen layout and uses fewer levels of hierarchy than is typical of graphical interfaces.

The VIF

The VIF is a program which is essentially an enhanced version of the X Window System terminal emulation window client *xterm*. VIF is normally configured for a given client Unix utility and one invocation of VIF is done for each utility started by Xgsh. The enhancements to *xterm* are provided in two major categories. The first is a set of mouse inputs that are translated into actions such as sending a character sequence through the pseudo terminal connection to the client utility. This permits the configuration of a certain amount of mouse input for a given client. The second major category consists of character stream filters for both input and output on the pseudo terminal connection. The filters are defined in the configuration using regular expressions (as in *Lex*) with an associated action similar to that related to a mouse input. This action permits certain visual interface widgets to be popped up, highlighted or sensitized upon matching of a given I/O character sequence. As such, this provides a limited facility for generating a graphical output translation of the character stream provided by the utility. For example, a compiler fatal error message might be translated to a skull and cross bones icon.

Mouse Inputs

The configured mouse inputs can be of several categories:

- selection on a configured visual interface widget such as a *pushbutton* or a *pull down menu*. The widget selection will then invoke the associated action. The widgets that can be configured are a button box for direct selection of buttons and a button box where the selections activate associated pull down menus.
- positioning of the text cursor via a left mouse button press on the text area subwindow. Internally, this generates the appropriate number of text cursor *up/down* and *left/right* arrow character sequences for the emulated terminal type and sends them down the pseudo tty connection to emulate the user pressing the arrow keys on a tty.
- a press of the center or right mouse button while the mouse cursor is on the text area subwindow. This then invokes the associated action for that mouse button. An example of where this can be useful is when the client utility uses menus that are selected with the arrow keys followed by a *<cr>* when the text cursor is at the appropriate item. By configuring an action of "send *<cr>* to client" for the right mouse button users can then do menu selection with the mouse.
- manipulating a scroll bar that generates scroll up/down actions. This scroll bar actually functions like a unidirectional joystick in that the thumb cursor returns to the midpoint as soon as the mouse button is released and generates some number of the up/down actions based on how far the thumb was displaced. (This has no relation to the *xterm* scroll bar.) This is primarily useful when the VIF is used as a front end to screen editors such as *vi* to allow scroll up/down operations to be generated using the mouse.

I/O Stream Filters

The Input and Output stream filters may be used as follows:

- The input filter can restrict or modify actions that the user may perform on the utility by translating the input key stroke sequence to an associated action which may include a replacement key stroke sequence. This can prove to be useful when a novice is being interfaced to a powerful (implies dangerous) utility such as *vi*. For example, the configuration could remove a control key stroke sequence from the input stream and display a graphical error icon.
- The output filter may allow output generated by the client utility to be translated that is more meaningful to the user. For example, "Bus error, core dumped" might be replaced by "Bad pointer reference during execution" or "Array subscript out of bounds, try using the Check option". The replacement action may include a graphical operation such as displaying an icon of a bell for the bell character ^G. The second style of action in the form of making an icon visible or sensitizing a widget for selection allows the configuration writer a limited use of graphical output for a client.

Actions

The actions invoked by the above can be configured to be any combination of the following:

- send a character sequence down the pseudo terminal connection to the client utility. This is the most common action and essentially translates mouse inputs to their keyboard equivalent for the client.
- send a character sequence to the terminal emulation subwindow. This must be done with caution since it leaves the terminal emulation subwindow in a state different from that assumed by the client utility. It may be used for effects such as reverse video or for message replacement if the client utility is not full screen oriented.
- change the visibility/sensitivity of the visual interface widgets. This permits a limited adaptation of VIF to the current state in which the client is functioning. For example, a screen editor in *insert mode*.
- pop up a text string or icon on the window. This action allows for a graphical translation of some of the client's outputs.
- change the mouse cursor momentarily or permanently. This may be used as a visual reflection of a change in the client utility's current state of operation.
- change the text cursor appearance. This may be useful as a visual reflection of the client performing some specific text operation.
- pop up a dialog box to request keyboard input. The box accepts a text string input from the keyboard that must pass a regular expression and then sends it down the pseudo terminal to the client. This can be used to enhance the input of a short text argument such as a file name or search string.
- pop up a yes/no query button box. The box has two command buttons for yes/no that send the "yes" or "no" strings down the pseudo terminal connection when the appropriate button is selected. This can be used to enhance the input of yes/no responses to verification queries.

Process Manipulation

The VIF also has certain standard capabilities for controlling the client process via mouse input. This is necessary so that process control may be handled in a manner analogous to that provided by the top level Xgsh. This includes:

- the window may be raised/lowered to ease use of several clients concurrently.
- Stop, Continue and Interrupt signals may be sent to the client to help control its execution. Depending on the configuration, the Stop/Continue signals may be implicitly generated upon the VIF window being raised/lowered.
- optionally text output may be page scrolled in a manner similar to *more* as an aid when the client is non-interactive, to avoid loss of output.

- the window may be configured to remain visible after the client process terminates to avoid the loss of any final output.

Problems and Limitations

The major limitations of the current VIF are its limited scope of graphical output and its very restricted knowledge of client utility state. A greater variety of graphical outputs for actions is needed to permit configuration builders to provide appropriate visual feedback. There is also a need to define some kind of client state knowledge in the configuration and relate client state to functional changes in VIF. Currently the only state information maintained is the visibility/sensitivity of the mouse selection widgets.

Another problem with the current VIF is that the I/O stream filters can easily hang the I/O stream if not very carefully configured. Any regular expressions requiring further characters to resolve the alternatives will hang the I/O stream until the characters are received. If the client is not doing further output or the user does not continue to type at the keyboard, the filter is hung.

The Xgsh

The Xgsh program is intended to replace a standard Unix shell as a top level command interface. It is a configurable visual interface where the items that may be mouse selected on the various panels are defined almost entirely by the configuration. A sequence of mouse selections are translated to the corresponding single line shell command as defined by the configuration. Once the command is constructed, it is run as a process subtree with a parent controlling window, usually provided by a VIF. The configuration file allows the configuration builder to create an appropriate set of commands for a particular user group and also permits a variety of restrictions to be enforced so that the novice will be protected from *dangerous* actions. For example, Pascal compiles might only allow text files ending in ".p" to be selected as file arguments. The screen layout (See Figures #2-5) is constructed of four panels that must be active for initial interaction. Other panels may then pop up to provide further interaction such as providing name arguments for commands. The main panels may be active at all times (as shown in the figures) or popped up when required from a menu bar, as defined by the configuration file.

Main Panels

The main panels allow selection of a top level (which defines a set of related operations), for the operations within these top levels, and also operations on directory tree. The top levels define a hierarchical structure that can be used to categorize operations into related groups. There may optionally be a directory associated with each top level to encourage use of a directory structure relating files to the operation groups. Selection of the current top level defines the group of operations that may be selected in the command panel and the current directory, if specified by the configuration. The directory tree operations panel exists as an independent entity to permit file name space browsing at any time. The only affect that directory operations have on command selections is the definition of the current directory. Along with these panels is a window that displays the current command under construction and can optionally allow text editing of the command.

Panel Layout

Each panel follows a standard layout that includes a variety of selectable items that are either text strings or icons. In some cases a scrolling widget or a dialog box for entering a text string at the keyboard may be included. The dialog box is used for getting *new names* or *search patterns* that are required via keyboard input.

Command Construction

Command invocation begins with a mouse selection of one of the operations in the command panel. If the associated Unix command line string (as defined by the configuration for that command selection) does not req a process window is created and the process subtree is forked off. Most often, the process window is provided by VIF but may simply be an icon window for clients that generate their own windows. If VIF is providing the window, the *stdio* of the associated Unix command line will be connected to VIF via a pseudo terminal. However, if name argument(s) are required, a name selection panel is popped

up to permit mouse selection of name arguments. Name arguments will most often be file names but may be other text strings as defined by the configuration. Optionally, a *new name* widget is included to allow keyboard input of new names as well as mouse selecti for existing ones. The name arguments are defined by the configuration in one of three ways:

- File names in a given directory that match a regular expression and criteria such as *text file* or *executable image*. An example would be text files matching *.c in the current directory, for a C compile operation.
- Names written to the standard output of a given Unix filter. For example, *sed "s/:*/|/" /etc/passwd* would generate all local user names which could be used as the name arguments fo operation.
- A list of names explicitly provided in the configuration file for the operation.

Once the name argument(s) are selected, the command can then be initiated by selecting a *Start* pushbutton widget. In cases where the operation is defined as only accepting one name argument and does not allow piping, the *Start* selection is not necessary.

The use of pipes has been a somewhat contentious issue, and is allowed to happen between certain operations as defined by the configuration file. Although powerful, pipelining operations can be dangerous and are somewhat at odds with the goal of a simple user interface. As such, the choice of whether or not to allow the use of pipes is left up to th builder.

The command being built is shown in the command string window either as the actual Unix shell command string being generated or as a chain of bubbles labeled for each command that is being piped together. The configuration can also allow editing of the command string in the window using the key-board for the case of actual Unix command line display. This permits a primitive facility to assist in the migration from the Xgsh interface to a standard Unix shell. This problem is discussed later.

Directory Operations

A set of operations on the directory tree are made available in a separate selection panel, since directory browsing tends to be a look aside activity. These operations may be interspersed with command synthesis to permit more flexibility during name argument selection. Command synthesis is only affected when a change in the current directory occurs. The directory operations are essentially a directory tree browser and provide the following functions:

- List the contents of the current directory.
- Change directory to the parent directory.
- Change directory to a subdirectory.
- Create a new directory.
- Delete an empty directory.
- Find names that pass a given regular expression in the current directory subtree.

The current implementation provides these operations as mouse selectable icons but work is in progress on a graphical representation of the tree to help traver directories and peek at their contents.

Configurations

The configuration files for both VIF and Xgsh are currently ".o" files compiled from large initialized C structures and the generation of a configurati is definitely in the domain of a Unix wizard. Even if a front end to the initialized structures was provided, constructing a configuration will require a good knowledge of Unix and the utilities that VIF is being interfaced to. The only component of the configurations that currently has a front end is the stream filter, which is defined by a Lex program.

Design Goals

The Xgsh design is aimed at providing a visual interface which is easy for a novice to use and may be tailored to a specific user group's needs. It is intended to provide another layer on top of the Unix pipeline of filters that protects the non-experts from the more *dangerous* aspects of the Unix command line, while allowing effective use of the system. In an effort to achieve this the idea of a configuration, to define

and restrict the user's environment, was developed so as to make the operations as simple and specific as possible. The user environment is meant to be sensitive to the user's current work context, as defined by the selected top level and operation, by remembering context information such as the name of the file that was most recently referenced. To make the visual interface as *natural* to use as was practical, a hierarchical structure which is context sensitive was devised and the number of mouse inputs required kept to a minimum. The mouse cursor is "warped" (ie. moved without mouse motion) frequently to follow sequence of the user. For example, if a "Pascal Compile" operation is selected, a panel with all Pascal source file names pops up with the mouse cursor at the source file name that was compiled the last time this operation was performed. This minimizes the need for the user to move the mouse and seems to reduce the frequency of the sensation of "Why do I have to tell it that again?".

Problems

A major problem with this type of environment is the difficulty of providing a migration path to more sophisticated shell interaction. A simple visual interface can never hope to be as powerful as a standard Unix `sh` but then how do you provide new capabilities for the user? The current Xgsh echoes the corresponding Unix shell command in a window and may allow the user to text edit the command under construction. This is a primitive attempt to address this problem, but not a particularly good one.

A possible approach to addressing this problem could be to include some sort of configuration extensibility facility to the Xgsh. In other words, a way must be provided by which users can modify or add new command operations to their configuration. This would allow them to further customize their environment and would act as a back door to teaching them the standard Unix shell, since the command's operation must be defined as a `csh` command line. Another simple solution is an escape to Unix shell facility that pops up a tty emulation window tied to an invocation of `csh`. It is felt that some facility of this kind is required to make the interface useful to the non-novice. In any case, this area definitely needs further investigation in order to provide an appropriate facility.

Another area that is not dealt with easily is the selection of file name arguments from different directories for the same command. The Unix file pathname syntax is far more powerful than mouse selection of file names created from the contents of a directory. As such, the selection of file name arguments that span several directories becomes difficult and cumbersome. A possible solution lies in the use of a directory subtree browser that displays a graphical representation of a subtree and allows mouse selection of directories to list for selection. If this is combined with the kind of selection criteria used by the Unix `find` command, it may be possible to build a fairly effective name selection tool. Unfortunately, this is not a simple thing to implement well and, even at its best, will be somewhat tedious to use when compared to a standard Unix shell.

Comparison with Existing Visual Command Interfaces

Current visual interfaces may be typified by a range in power and complexity extending from the Apple Macintosh to the Xerox Interdisp-D environment.

The Macintosh environment is based on the desk top metaphor and the concepts of applications and associated documents in file folders. The file folder essentially defines a set of user programs (applications) and the documents and as such enforces a partitioning of the file system based on the various file folders. This is vaguely similar to the Xgsh top levels when each top level is configured to use a separate directory subtree. The Macintosh *Finder* is used to select a current application and is in some sense akin to command selection on Xgsh. However, Unix and therefore Xgsh, does not define any structural relationship between an application and its associated documents. The Xgsh allows a configuration to define a directory for each top level, which does provide the capability to construct a somewhat similar hierarchy but does not enforce such a structure on the configuration designer. The standard menus on the Macintosh provide certain standard operations, such as *save* on the File menu and are an effort to create a consistent interface. Xgsh does not have any similar concept, except possibly the directory operations panel, and leaves the operations definitions up to the configuration designer. On Xgsh the name selections are done in response to operation invocations whereas on the Macintosh a selection of the *open* operation on the File menu would be required. The Xgsh design allows the configuration designer to use either icons or text strings for selection items in most cases except file names, whereas the Mac is heavily icon oriented. In

general, the Macintosh provides a simple consistent interface, but one that enforces a structure that is not inherently a part of Unix. As such, this kind of structure was avoided in Xgsh, leaving a maximum of leeway to the configuration designer.

The Interlisp-D environment is an integrated system intended for an expert programmer in LISP. It makes essentially all of the system internals available to the programmer in an environment where the many tools cooperate in complex ways. As such, the interface is powerful, flexible and very confusing for the novice. Since Xgsh was designed with the novice in mind, this aspect of Interlisp-D was considered inappropriate. However, several aspects of the Interlisp-D environment were considered highly desirable. One area where Interlisp-D shines is its extensibility. As the user brings new LISP functions into the system it really does become their *personal* computing environment. As mentioned before, although there is a great potential in extensibility, it is not a part of the current Xgsh. Another aspect of Interlisp-D is its presentation of a great deal of information on system activity to the user, often in a graphical form such as changes to the mouse cursor. This has been incorporated into Xgsh to the limited extent possible within the confines of Unix. The actual top level commands on the Xerox Interlisp-D environment are still lisp functions invoked via. key stroke input. However, there are several graphical components used to assist the user during command execution, some of which has influenced the Xgsh design. One such assist is a file browser that is somewhat similar to the subtree browser planned for the directory operations panel. Another borrowed feature is the process menu that Interlisp-D provides to permit users to probe and control processes. To summarize, the Interlisp-D environment was at a level of complexity that we wished to avoid but provides features that influenced the current design and future directions.

User Experience

Unfortunately, due to delays in the installation of a workstation laboratory on our campus, user experience has been limited to the developers and short demonstrations to outside parties. However, the design was based on a preliminary *gsh* that provided a menu based interface for students working on ASCII terminals. The experience with this *gsh* was promising in that our novice students quickly adapted to the environment. In fact, it was observed that within a few hours use many students would type ahead menu selections before the menus were displayed. Although this menu shell was intended strictly for introductory level students, many continued to use the interface for several semesters and found the transition to *csh* difficult. The VIF and Xgsh software is at the stage where a fairly extensive user evaluation is required to refine the design.

Directions

There are several areas that need to be addressed in an effort to enhance the G shell environment. As mentioned above in the Problems section, the extensive and more flexible configurations are both required to cover a wider spectrum of the user community.

There is a requirement of a framework for defining inter window/process interaction. It seems appropriate to allow the definition of stream connections with VIF style filters that interconnect the *stdio* of various VIFs and their client processes. This should be doable at the Xgsh level, since it fabricates the VIFs and the clients processes during command invocation.

The maintenance of an AI knowledge base of recent activity and an associated rule base to define default next selections would provide some measure of state sensitivity and help to minimize the requirements for mouse inputs. Minimizing mouse input is important, since the sophisticated user rapidly tires mouse movements. As a first step, the provision of hooks into an AI software system is required to help explore the use of expert assistants. This type of support could be incorporated at both the Xgsh and VIF level interfaces, but requires a great deal of forethought for there to be a reasonable chance of success.

Summary

Although the software described in this paper is only a preliminary step, it does seem possible to build visual interfaces for shell level interaction on a Unix workstation. The environment described in this paper provides a simple interface to Unix configured to a specific user group's needs. But, in doing so, it provides only a small subset of the Unix command interface functionality tailored to that user group. Critical shortcomings of the current interface come in the areas of extensibility, inter window/process

interaction and knowledge of recent activities. Without these capabilities, more sophisticated users rapidly outgrow their configured visual interface and often find it tedious. As such, the current G shell environment for Unix is probably most appropriate for the novice or casual user.

Acknowledgements

We wish to acknowledge the information and assistance provided by the Digital Equipment Corporation, without which this software might never have been developed.

References

- A. Averill, *Macintosh User Interface Guidelines*, Apple Computer Inc., 1984
- S. Draper, "Display Managers as the Basis for User-Machine Communication", *User Centered System Design*, 1986
- J. Gettys, et al., "Xlib - C Language X Interface", *MIT Project Athena*, 1988
- J. Gettys, "Problems Implementing Window Systems in UNIX", *Proceedings of USENIX User Group Conference*, January, 1986
- Daniel Gill, "A Proposal for Interwindow Communication and Translation Facilities", *Proceedings of USENIX User Group Conference*, January, 1986
- Hewlett-Packard Company, *Programming With the X Window System*, 1986
- Stephen N. Kahane, et al., "Windows in the Hospital", *Proceedings USENIX User Group Conference*, January 1986
- Thomas Neuendorffer, "GLO - A Tool for Developing Window-Based Programs", *Proceedings USENIX User Group Conference*, January 1986
- R. Reichman, "Communications Paradigms for a Window System", *User Centered System Design*, 1986
- R.W. Scheifler, "X Window System Protocol, Version 11", *MIT Project Athena*, 1988
- Sun Microsystems, *Windows and Window Based Tools: Beginner's Guide*, February 1986
- W. Teitelman and L. Masinter, "The Interlisp Programming Environment", *IEEE Computer*, April 1981

X11/NeWS Design Overview

Robin Schaufler

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
robin@sun.Com

Abstract

The X11/NeWS server is a general, unified window system server that supports two front-ends, one for the X11 protocol and one for the NeWS protocol. Because the X11 semantics and the NeWS semantics share a great deal in common, much of the underlying implementation is common to both. Where the differences cannot be resolved to a common model, the union of the two sets of semantics is provided. By allowing both protocol implementations to coexist in a single server, both X11 and NeWS clients can share the screen and the input devices. A single window manager that manages windows created using both protocols presents an integrated interface to the user.

Introduction

The X11/NeWS server is a merge of two window systems: the *X Window System, Version 11**[1] and *NeWS: the Network Extensible Window System**[4]. X11 is an emerging windowing standard, developed at MIT. NeWS is a window system developed by Sun Microsystems, Inc., which is based on the PostScript* language[6], an emerging standard in page description languages, developed by Adobe Systems, Inc.

The goal of the merge of X11 and NeWS is to produce a single server process that supports the entire semantics of both the X11 and NeWS protocols, is portable to a wide variety of hardware, and supports portable X11 extensions. Also, it is important that a single window manager can manage all windows on the screen regardless of which protocol is used to create them, thus presenting an integrated interface to the user.

A window manager that uses NeWS protocol can manage windows that are owned by either X11 or NeWS clients. In contrast, although window manager written to use X11 protocol can manage windows owned by X11 clients, it can only manage windows owned by NeWS clients that follow the rules of the X11 protocol.

The X11 protocol specification (see [1]) defines a means of extending the X11 protocol. The MIT sample server implementation provides a mechanism for implementing extensions, documented in the *X11 Server Extensions Engineering Specification*[3]. While the latter mechanism is still under development, the intention is that extension implementations that use it will be portable across X11 server implementations. X11/NeWS supports portable X11 extensions.

The extension specification requires that a small set of include files and the source to one procedure be provided with the server. Without any other source, an extension supplier should be able to recompile a portable extension with the X11/NeWS include files, link it with X11/NeWS object libraries, and have the extension work.

* X Window System is a trademark of MIT
NeWS is a trademark of Sun Microsystems, Inc.
PostScript is a registered trademark of Adobe Systems Inc.

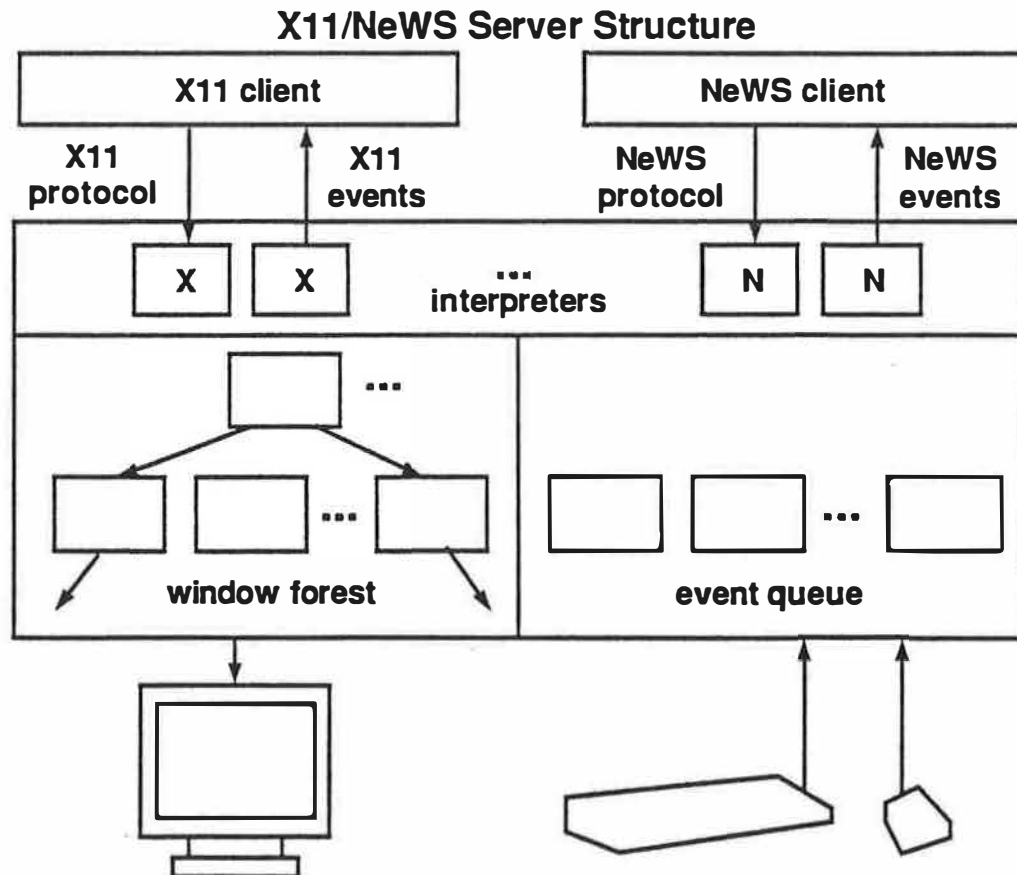
Architecture

There are three major functions that a server for either protocol must perform:

1. Scheduling interpretation of protocol
2. Allocation of portions of the display
3. Distribution of input

Below is a figure illustrating an architecture that provides these functions for both X11 and NeWS, which shows the structure of the X11/NeWS server. If the X11 client and interpreter were omitted, this figure would illustrate the structure of the existing NeWS server. If the NeWS client and interpreter were omitted, this figure would illustrate the structure of the X11 sample server from MIT.

In the figure below, the boxes labelled **X** represent interpretation or generation of X11 protocol, and the boxes labelled **N** represent interpretation or generation of NeWS protocol. The boxes in the area labelled **window forest** represent *windows* (called *canvases* in NeWS), which are portions of the screen on which clients can draw. The windows in the diagram are unlabelled because the protocol which was used to create them is irrelevant to the structure of the window forest and to the structure of the windows themselves. The boxes in the area labelled **event queue** represent events, and are likewise unlabelled because the protocol used by the recipient of the events is irrelevant to event queuing and distribution.



The Scheduler

Given multiple clients of a window server, a way must be provided to fairly and predictably schedule execution of requests from each. Allocation of time between multiple clients is required of all X11

servers, and all NeWS servers; therefore it is also required of the X11/NeWS server.

NeWS schedules between lightweight processes (*lwps*). A process is a thread of control; a lightweight process is a process which shares its address space with other lightweight processes. In NeWS, a context switch may occur when a lwp blocks or explicitly gives up control. In contrast, X11 schedules between clients. In X11, a context switch may occur between any two requests, unless a client has grabbed the server.

An X11 client is a source of a sequence of X11 requests. A sequence of X11 requests is a linear thread of control. Since a linear thread of control is a subset of the possibilities offered by a process, the X11/NeWS server represents X11 clients as lightweight processes.

Context switching and scheduling are entirely internal to the X11/NeWS server and are in no way dependent on the operating system. An X11/NeWS lwp is represented as a context structure, whose contents include the protocol interpreter associated with that lwp, and the source of protocol to be interpreted. The source of protocol may be a client connection or downloaded code.

Running a lwp means calling the associated interpreter procedure and passing in the context structure for the lwp. The interpreter checks a single state field in the context structure to resume where it left off when it last gave up control. As it executes, the interpreter maintains state in the context structure. To give up control, the lwp saves a value in the state field that it checked earlier when it resumed execution, and returns to the scheduler.

The scheduler makes no distinction with regard to the protocol interpreter associated with a lwp. A lwp runs until it gives up control, and is run again when it is ready.

Sharing a Screen

NeWS and X11 are very similar in the way that they treat the screen. Both allocate portions of the screen on which a client can draw, called *canvases* in NeWS, and *windows* in X11. From here on, these terms will be used interchangeably. Both X11 and NeWS permit unlimited nesting and overlapping of windows. Both provide for expression of interest in events occurring on specific windows, including damage (exposure) and device input.

The X11/NeWS server has a tree of nested windows for each screen; the aggregate of these trees is called a forest. The server allows the cursor to roam across screens, in some device dependent geometry. An extension to the NeWS protocol for getting the list of screens is offered.

The structure underlying a node in the forest is called a *canvas*, for historical reasons. The forest of canvases in the X11/NeWS server is homogeneous; the same canvas structure is used to represent both canvases created from NeWS protocol and windows created from X11 protocol. Since the forest is homogeneous, if a canvas is reconfigured to expose regions of underlying canvases, damage will be detected on all of them whether they were created using NeWS or X11 protocol.

NeWS canvases and X11 windows do have some differences. They provide different attributes, imaging models, color models, and font models. A more detailed description of how these differences are resolved in the X11/NeWS server is given below.

Properties

Each X11 window has a property list. NeWS canvases do not currently have property lists but will benefit from the addition. Canvases appear to NeWS programs as PostScript dictionaries; fields of a canvas appear as keys in the dictionary that represents it. NeWS programs can access the property list for a canvas through a new key in the canvas dictionary.

X11 Properties are quadruples: *name*, *type*, *format*, *data*.

- *name* and *type*

Both of these elements are X11 atoms, which map well onto the PostScript language name *type*[6].

- *format*

The format describes the unit size of *data*, which can be 8, 16, or 32 bits.

- *data*

This element consists of data that is handled but not interpreted by the server. It is represented by a NeWS string, which is essentially an array of 8-bit bytes. 16 and 32 bit format are implemented by having a string that is a multiple of 2 or 4 bytes long.

A *property* is constructed by combining the above elements into a four element array of in the order of name, type, format, and data.

A *property list* is an unordered set of properties. The most convenient data type in PostScript language for unordered sets is a dictionary, which is essentially a hash table. The property name serves as the key to a property.

Display Attributes

While X11 and NeWS have many display attributes in common, there are a number of display attributes that are only accessible through one protocol or the other.

For example, X11 windows have borders, background, and gravity. These attributes allow an X11 server to immediately tidy the screen after a change to the window hierarchy without the overhead of a server-client round trip. In NeWS, downloaded code is used to perform those display update functions that need instant response, and therefore need to avoid the overhead of a round trip; therefore, NeWS does not provide these attributes. In X11/NeWS, these attributes are implemented by enhancing the canvas data structure and operations.

For another example, NeWS protocol provides access to a canvas' arbitrary shape, which is not necessarily rectangular. As X11 core protocol defines only rectangular windows, it only provides access to a canvas' bounding box. If an X11 client gets a handle on a non-rectangular canvas and inquires about the size and origin, it receives the bounding box of the canvas in the reply. If an X11 window is exposed by an operation on an occluding non-rectangular canvas, the exposed region is approximated by some number of rectangles to the resolution of the display.

Colormap Access

While NeWS attempts to hide the details of the workstation's display hardware from clients, X11 attempts to describe them. The X11 connection setup protocol specifies the pixel depth and colormap models available for each screen on the workstation in terms of *visual types*. One screen may have more than one visual type. For example, a framebuffer containing 8-bit deep color and an overlay plane offers a single screen with two visual types, one 8-bit PseudoColor, and one 1-bit StaticGray.

Each visual type supports the request `CreateColormap`. For static visual types (StaticColor, StaticGray, and TrueColor), `CreateColormap` returns the single static colormap provided by that visual type. For dynamic visual types (PseudoColor, GrayScale, and DirectColor), it returns a newly-created colormap. Each visual type has a default colormap, whose initial population of colors is not defined in the X11 protocol specification.

For devices that offer a hardware colormap, the X11/NeWS server populates part of the default colormap with read-only, sharable colors in the form of a colorcube whose axes are red, green and blue. Doing so allows NeWS and X11 applications that do not require dynamic colormap access to share the colorcube. X11 applications needing dynamic colormap access either allocate colors from the rest of the default colormap, or allocate their own colormaps. An extension to NeWS provides dynamically modifiable colors in the rest of the colormap as well.

Each screen has some number of colormaps installed. When a colormap is installed, primitives drawn using pixels assigned from that map appear in their correct colors. When any application installs its own colormap, and the combined set of allocated colors overflows the hardware colormap, other applications (both NeWS and X11) appear wrong until their colormaps are reinstalled. The incorrect appearance is a problem in any implementation of X11 colormaps.

Imaging Model

NeWS is based upon the stencil-paint imaging model offered by the PostScript language. The stencil-paint imaging model reveals no notion of pixels, and allows a PostScript program to set an arbitrary 2-D coordinate system (called world coordinates), whose units may be fractions of pixels. Even the default coordinate system might not be in units of screen pixels, as the PostScript language defines the default units to be 1/72 of an inch. Due to the arbitrary coordinate system, neither the stencil-paint imaging model nor the PostScript language make any guarantees about which pixels are touched when rendering the graphics primitives.

Raising the level of abstraction from pixel coordinates to world coordinates has some advantages. For example, if an image is described in world coordinates, switching from a low resolution to a high resolution device affects only image *quality*, not image *size*. Yet pixel coordinates are still accessible by setting the transformation matrix to be the identity matrix. Deferring the transformation of world coordinates into pixel coordinates to the server allows the server to utilize matrix multiplication hardware. By making no guarantees about which pixels to touch when rendering an image, the server can utilize of polygon and vector rendering, and anti-aliasing hardware.

In contrast to NeWS, X11 specifies a pixel based imaging model. Pixel coordinates are used throughout the X11 drawing operations. With the exception of *narrow* lines (i.e. lines specified to have a width of zero), the X11 protocol specification describes rendering algorithms for all drawing primitives that guarantee pixel accuracy.

The X11 imaging model can be thought of as a precise specification of the pixels involved in drawing operations when the current transformation matrix is the identity matrix. If a graphics accelerator uses a different rendering algorithm from the algorithm specified by the X11 protocol, the X11 protocol specification requires the server to bypass the accelerator. If the server device drivers ignore the accelerator, both X11 and NeWS drawing operations can use the same rendering algorithms without loss of correctness or precision.

Operations defined in the X11 protocol but not in NeWS include *rasterops*, *tiling*, and *stencilling*. Similar operations are needed to implement the PostScript language imaging model on a bitmap display; they have been enhanced to support the X11 imaging model in the X11/NeWS server.

Fonts

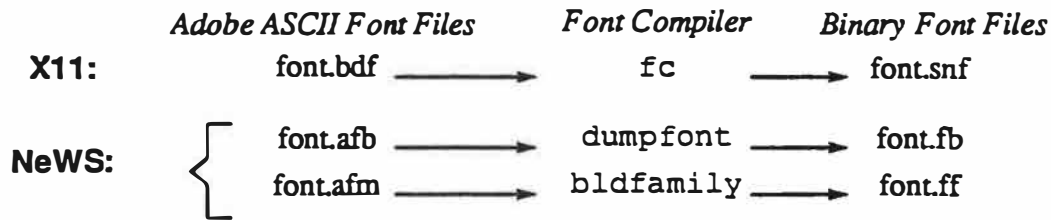
Both NeWS and X11 currently support fonts described in Adobe Character Bitmap Distribution Format (*bdf*), although X11 fonts are based on a more recent version of *bdf* (2.1). These fonts are provided as ASCII files containing information about the family and face of the font followed by specific information and bitmaps for each of the characters in the font. NeWS and X11 each have a mechanism for converting these ASCII font files into a machine-dependent format suitable for efficient processing by the window system server. The diagram below shows all the font files and utilities.

In the case of NeWS, ASCII font files are pre-processed by the *dumpfont* utility to create binary font files consisting of a font structure followed by an array of glyph structures. For each font family, there also exists an ASCII font family description file which is pre-processed by the *bldfamily* (build-family) utility to create a binary font family file. NeWS handles outline and stroked font ASCII formats as well as bitmap fonts.

X11 uses a similar scheme to pre-process ASCII font files for subsequent loading in response to client requests, using the *fc* font compiler utility. They consist of data structures containing font information, the glyphs themselves, and a number of font properties.

In X11/NeWS, both NeWS and X11 font requests are processed through an enhanced version of the existing NeWS font machinery. In the case of X11, font and glyph information, as derived from binary font files and maintained in NeWS data structures, are mapped to X11 data structures to be passed to the X11 interpreter as font requests are serviced. The X11/NeWS font file format and font compiler are upgraded to Version 2.1 of Adobe's Character Bitmap Distribution format, and enhanced to

include X11 font information.



Font files and compilers

X11 assigns no format rules for font names, although it does follow certain naming conventions for groups of fonts. From the font name, the OS dependent code generates the font file name for the file that contains the font. Any incompatibilities between NeWS-style and X11-style font names and font file names are resolved there.

The Event Queue

The primary difference between the X11 input model and the NeWS input model is that NeWS event distribution consists of message passing between lightweight processes within the server, whereas X11 event distribution consists of sending information to external clients. In the following discussion the term *client* is a lwp in NeWS and a client-side program in X11.

In spite of the major difference in input models, there are many points on which NeWS and X11 agree:

- events are collected from devices and timestamped as quickly as possible.
- events from all devices are *serialized*.
- events are normally delivered to clients in the order generated. Exceptions are X11 grabs and NeWS events whose timestamps are not the same as the time at which they were generated.
- clients indicate classes of events they wish to receive.
- events propagate up the window tree, but may be stopped explicitly by windows. Exceptions are X11 grabs and NeWS global interests.
- focus for keyboard events can be set explicitly.
- default mouse and keyboard input distribution follows the mouse.

Additionally, there are areas where X11 event distribution is a subset of NeWS event distribution.

- NeWS events are perceived by PostScript programs as dictionaries; one event dictionary can carry all the information needed for X11 events.
- NeWS interests are templates of the events or sets of events that match them. Such a template may be constructed for any set of events corresponding to any X11 interest mask bit.

Delivering Events to X11 Clients

In NeWS, PostScript programs form a bridge between receipt of a NeWS event and delivering the information in that event to an external client. A PostScript program extracts relevant information from a NeWS event and writes it to the appropriate external connection.

In X11/NeWS, events are delivered to an X11 client by a lwp that waits for a NeWS event, then writes it in the form of an X11 event to its connection. As NeWS provides no mechanism for a lwp to block on two things at once, the lwp that blocks on a NeWS event is not the same one as the X11 interpreter lwp, which blocks on a connection read. The lwp that waits for events is called the *input agent*. The input agent is a NeWS program that runs in the server, but calls X11-specific primitives.

The conversion of NeWS events to X11 events has several advantages. First, an event may be enqueued with no notion of how it will ultimately be distributed. Second, both NeWS lwps and X11 clients can receive the same events, improving the overall integration of the system.

Given that events are distributed to the input agent lwp, rather than the X11 interpreter lwp, interest in those events must be expressed by the input agent lwp. However, the X11 interpreter lwp receives the requests that set the client's interest. To reconcile this, the X11 interpreter lwp keeps a pointer to the input agent lwp, and adds interests to the input agent's interest list.

Event Distribution Within the Server

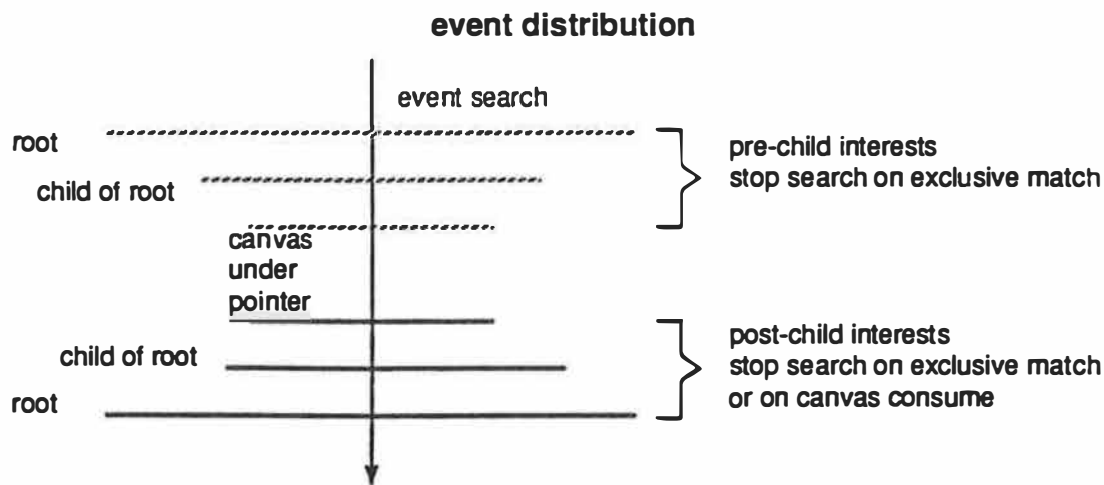
In NeWS, a lwp expresses interest in receiving particular events by creating a template event, known as an interest. Event distribution consists of searching the interests, using them as patterns to match the event against. For each interest matched by the event, a copy of the event is placed on the local queue of the lwp that expressed the matched interest. The NeWS `awaitevent` operator returns the head of the local queue.

An event may contain a canvas. In NeWS, when an event is expressed as an interest, if it contains a canvas, it is placed on that canvas' interest list; otherwise it is placed on a global interest list. The canvas hierarchy determines the order of the interest search during event distribution. When an event is distributed, if the event contains a canvas, only interest list for that canvas is searched. Otherwise, first the global interest list is searched, then the interest list of each canvas from the canvas under the mouse to the root is searched. The search is terminated

- If an exclusive interest is matched
- If an interest is matched on a canvas that consumes matched events
- If a canvas whose interest list was searched consumes all events.

The bottom-up distribution rule corresponds well to X11 device input distribution for most cases. The global interest list serves active grabs well. However, the rules for X11 passive grabs (described later) really correspond better to top-down distribution. In order to achieve desired performance given the possibility of passive grabs, the event distribution rules have been generalized in X11/NeWS.

In X11/NeWS, each canvas has two interest lists, called *pre-child* and *post-child* interest lists. To distribute an event that specifies a canvas, the server searches the pre-child interest list of each canvas from the root to that canvas, then searches the post-child interest list of that canvas. To distribute an event that does not specify a canvas, the server searches the pre-child interest list of each canvas from the root to the canvas under the mouse, then searches the post-child interest list of each canvas from the canvas under the mouse to the root. Only a match of an exclusive interest can terminate the search on the way down. The old rules for terminating the search still apply on the way up. The global interest list is replaced by the pre-child interest list on the root canvas. The diagram below shows the distribution process.



The application of this general event distribution scheme to the various X11 input concepts is described below.

High Level X11 Concepts

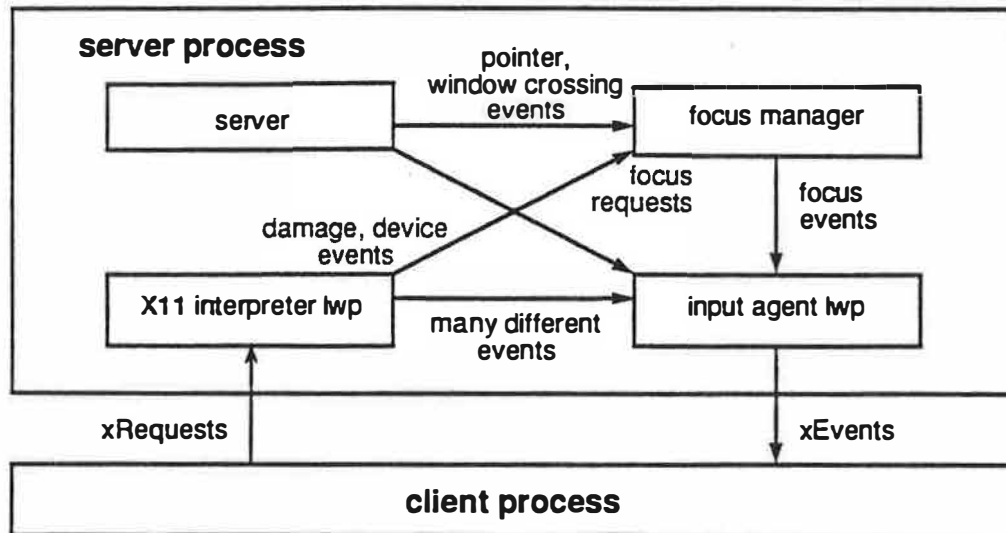
The X11 protocol includes a number of concepts that are omitted from NeWS, as NeWS was created with the expectation that similar capabilities would be implemented as PostScript programs downloaded to the server. In fact, NeWS is currently distributed with a set of PostScript programs that implement most of these capabilities. These PostScript programs are not distinguished from other PostScript programs by the server, but they are special in that they perform functions that are generally considered to be system functions.

Examples of functions included in the X11 protocol but not built into NeWS are:

- Keyboard Focus
- Grabs
- Selections
- Cursor confinement to a window
- Shift-modifiers, key mapping, and mouse button mapping

One of the PostScript programs that provides system functions is called the *focus manager*. It manages the keyboard focus according to the focus model chosen by the user. There is only one incarnation of this program at any one time. The focus manager is responsible for maintaining the current input focus, and for sending FocusIn and FocusOut events when the focus changes.

Below is a diagram of the flow of events among the server, the focus manager, the X11 interpreter lwp, and the input agent lwp. The diagram leaves out some details in the interest of clarity. Arrows within the server process denote source and destination of events, not control flow. Hence, the centralized event queue is not depicted. Arrows between server and client processes denote source and destination of interprocess communication. Both of these arrows represent the same network connection; they are only shown separately to distinguish the flow of data. The focus manager also distributes events to NeWS lwps, but this is not shown. Event types are representative, not all-inclusive.



Focus

In NeWS, lwp's that want keystrokes to go to particular canvases register those canvases with the focus manager. These canvases are called focus clients. When a focus client is notified of getting the focus, it expresses a pre-child interest in keystrokes on the root. When it is notified of losing the focus, it revokes that interest.

In X11, the focus is not the destination of keystrokes; rather it is a ceiling on bottom-up keystroke distribution. That is, if the focus window does not contain the mouse, then only the focus may get the keystrokes. However, within the focus window, keystrokes are distributed bottom-up. The focus window itself need not be interested in key presses or releases.

In X11/NeWS, when a window owned by an X11 client gets the focus, its input agent expresses an exclusive pre-child interest in keystrokes on the root, and also expresses an exclusive post-child interest on the focus window. When it loses the focus, it revokes those interests. When an event matches the first interest, the input agent looks in a dictionary maintained by the focus manager to see whether the focus contains the pointer. If the focus does not contain the pointer, the input agent specifies the focus canvas in the event; otherwise it specifies no canvas in the event. The input agent then redistributes the event, which may match X11 key press and release interests. If there are none, the interest search is terminated by the exclusive post-child interest on the focus window.

Active Grabs

X11 allows a client to actively grab the keyboard or the pointer. Events from the grabbed device always go to the grabbing client. A grab may also specify synchronous or asynchronous mode for either or both devices. Synchronous mode for a device results in freezing that device until either a request explicitly thaws it or an event implicitly thaws it.

The input agent for a grabbing client expresses pre-child interest on the root canvas for the device it is grabbing and for the other device if the grab specified synchronous mode. When an event matches this interest, the input agent calls an X11-specific operator that buffers events for frozen devices, checks pointer events against the pointer grab event mask, translates grabbed events into X11 events, and delivers them to the grabbing client.

Passive Grabs

An X11 grab specified for a key or a button is a passive grab. A passive grab is specified relative to a grab-window. Pressing a button activates the matching button grab on the highest grab-window between the root and the canvas under the mouse, if any exist. Pressing a key activates the matching key grab on the highest grab-window between the root and either the focus or the canvas under the mouse, depending on whether the focus contains the mouse.

A passive grab is expressed as a pre-child interest on the grab-window. Pointer events are always sent with no canvas specified in the event. Therefore, the highest pre-child interest in the ancestry of the canvas under the pointer matches, so the highest button grab gets the event. Keyboard events will always match the pre-child interest on the root corresponding to the focus. However, when the event is redistributed, the pre-child interest search is continued, so the highest key grab above either the focus or the canvas under the mouse gets the event.

When an interest for a passive grab is matched, the input agent calls an X11-specific operator which caches the event for replay, activates the grab, and goes through active grab processing.

AllowEvents

The X11 AllowEvents request has many modes. The replay modes redistribute the cached event that activated the grab, giving lower passive grabs a chance to match. The other modes resend buffered events, replacing them on the event queue. The event queue is ordered by timestamp. The timestamps of the buffered events are left untouched, so they are likely to be the earliest on the event queue. If the AllowEvents mode was synchronous, the first of these resent events to match the active grab will refreeze the device, starting the buffering all over again.

Crossing Events

X11 keyboard focus and window border crossing events include more information about the relationship between the windows that the focus or cursor is exiting and entering than do analogous NeWS events. The pointer crossing code in X11/NeWS has been enhanced to put this additional information in the crossing events. In response to getting a focus event, the input agent calls an X11-specific oper-

ator, which goes through the pointer crossing code to determine the details to send the X11 client.

Selections

In general, there are two popular selection models found in various window systems: the request model, and the buffer model. In the request model, a selection service keeps track of the clients holding the various classes of selection. When a client requests information about a class of selection, the service passes that request on to the holder of the selection, or returns some means by which the requestor and the holder can communicate directly. In the buffer model, also known as the clipboard model, the entire contents and attributes of a selection are transferred to the selection service, which answers requests about selections directly. The request model is more general, and handles huge selections better than the buffer model, but at the cost of client complexity and interprocess communication overhead.

X11 offers both models, provided that a convention exists whereby a client that wishes to inquire about a selection can determine which model is in use. NeWS offers a selection service implemented as a PostScript program, which provides both models transparently to an inquiring client. To preserve NeWS' ability to support both models transparently, the selection service continues to be implemented as a PostScript program.

In X11/NeWS, the X11 interpreter implements SetSelectionOwner, GetSelectionOwner, and ConvertSelection requests by sending an event to the input agent, which then goes through the standard NeWS selection interface. In NeWS, set and get the selection owner are implemented by setting and getting values in a dictionary. When an X11 client becomes the selection owner, its input agent acts as the selection holder on its behalf. To implement X11 ConvertSelection requests, the input agent looks in the dictionary for the requested selection. If the requested selection does not exist, then the input agent delivers an X11 SelectionNotify event with property None to its client. If the selection exists, but is buffered, then the input agent delivers an X11 SelectionNotify event with the requested contents to its client. Otherwise, the input agent sends a NeWS SelectionRequest event to the holder, which may be the input agent of an X11 client. The holding input agent converts the NeWS SelectionRequest event to an X11 SelectionRequest event and delivers the event to the holding client. It is the responsibility of the holding client to send an X11 SelectionNotify event to the requestor. Note that because NeWS and X11 share the selection mechanism, cut and paste between NeWS and X11 applications is possible.

Since an X11 selection is always associated with a window, but a NeWS selection does not have to be, an X11 GetSelectionOwner request may return null even if a selection exists. However, X11 ConvertSelection requests still return useful information, so this discrepancy does not seem to be worth resolving.

Confining the Cursor to a Window

NeWS has no such notion; X11 requires it. The X11/NeWS server cursor code is enhanced to implement this function for X11 only.

Shift-modifiers

X11 clients cannot set the state of a shift-modifier, but NeWS clients can, which results in events being sent to X11 clients as if the hardware modifier really changed state.

The set of X11 modifiers is restricted to 8, but NeWS clients may use more. The first three modifiers have a globally defined meaning for X11: Shift, Lock and Control. Names for shift, lock, and control are provided so that NeWS clients may interpret these modifiers the same way as X11 clients.

Note that by expressing interest in both key presses and releases, and by inquiring the state of the keyboard on FocusIn and EnterNotify, an X11 client may keep track of the state of any keys, and therefore may interpret any number of keys as modifiers. However, only the state of the 8 modifiers is reported with each key press or release, and only the 8 modifiers may be used to qualify key grabs.

Memory Management

NeWS offers a garbage collected implementation of the PostScript language. When a lwp creates an object, it gets a **direct reference** to it. When an object ceases to be referenced, it is destroyed and its memory reclaimed. X11 uses a resource database to maintain an extra level of indirection. When an X11 client creates an object, a reference to it is stored in a resource database indexed by resource ID, and subsequently the ID is passed around. If the object is destroyed and its memory reclaimed, its resource ID will then refer to null. As long as the implementation of a request checks for null return values from resource ID look-ups, this is a robust approach.

In the X11/NeWS server, the resource database is implemented as a hierarchy of PostScript language arrays and dictionaries. To free resources after a client dies, the dictionaries for the client's resources are removed from the resource database, normally causing them to cease to be referenced, and thereby destroyed. However, another NeWS lwp, most likely a system lwp such as the focus manager, may have gotten a valid reference to some objects in the resource database. In order to allow these other references to be flushed, the notion of a soft reference is introduced. An operator is provided which takes a reference to an object from the operand stack and replaces it with a soft reference to that object. When the only remaining references to an object are soft, an Obsolete event is sent. Any lwp that sets references to an object to be soft should express interest in the Obsolete event for that object, and should respond to a match on that interest by flushing all its references to the object.

X11 objects that may be interesting to NeWS lwps are stored in the resource database as NeWS objects. For example, window and pixmap resources are represented by canvases. X11 objects that are uninteresting to NeWS lwps or that do not map to NeWS object types, notably extension objects and fake objects, are stored in a new type of NeWS object, called the opaque type. An opaque object may be pushed and popped on and off the stack, saved in dictionaries and arrays, etc. but not directly manipulated in any way.

NeWS lwps are not required to associate resource IDs with the objects they create. But since the resource database consists entirely of PostScript language objects, a NeWS lwp has the option of creating resource IDs and associating its objects with them. By doing so, it makes its objects accessible to X11 clients. This is especially handy if a NeWS canvas might be managed by an X11 window manager. If a NeWS lwp creates an object with no resource ID, no X11 client can ever find out about it.

Connection Management

In NeWS, the loop that listens for new connections is implemented as a PostScript program, which blocks, waiting for a new connection. When a connection occurs, the listening lwp forks a new lwp, which executes the file representing the connection. An X11 connection is implemented the same way, except that instead of executing the file, the listening lwp forks two more lwps: the input agent and the X11 **interpreter**. The input agent is written as a PostScript program; it sits in a tight loop waiting for NeWS events, converting them to X11 events, and sending them to the X11 client. The X11 interpreter process is like a NeWS interpreter process except that instead of calling the PostScript **exec** primitive on the connection file, it calls a new primitive, **xinterp**. The **xinterp** primitive substitutes the X11 interpreter for the NeWS interpreter and then proceeds to execute the requests coming in on that connection. When the **interpreter** lwp dies, its parent lwp kills the input agent, cleans up, and kills itself.

Authentication

It is currently possible for the PostScript program that accepts connections to discover the name of the host the connection originated from, and to complete the connection only if that host is in a dictionary of authorized hosts. This is the level of protection offered in X11. The X11 access control requests manipulate the dictionary of authorized hosts.

Reset

The X11 protocol specifies that the server is completely reset whenever the number of its clients

goes to 0, and the last client disconnects with mode set to Destroy. However, in NeWS, many lwps may be active without the benefit of a connection. These lwps are treated as having open connections, and some never die. For example, if the lwp that listens for connections died it would be impossible to connect to the server, therefore at least that lwp must survive.

Problem Areas

For X11/NeWS to be completely successful, it should function as if it were a standalone X11 server, and also as if it were a standalone NeWS server, yet allow clients of both protocols to function in perfect harmony with each other. However this is not entirely the case.

Impact on X11

X11 clients do not see any difference between the X11/NeWS server and a standalone X11 server. Requests are interpreted exactly according to specification, events are reported exactly according to specification. However, NeWS lwps that do not procure client IDs and associate resource IDs with the canvases that they create effectively hide canvases from X11 query requests. Furthermore, it was considered unnecessary to implement server reset on final client death, as some lwps never die. No X11 client can ever detect that the server does not reset, but it does not.

Impact on NeWS

As a side effect of the work that needed to be done to support X11, some new capabilities have been added to NeWS.

- The color model for NeWS has been extended to provide modifiable color objects.
- The underlying graphics system generates visibility, gravity, and unmap notification in addition to damage notification.
- Pre-child interests replaced global interests.
- Window crossing and focus events contain more information than they used to.
- The notion of soft references has been added.

There is a minor incompatibility introduced by pre-child interests. In standalone NeWS, if a canvas was specified in a sent event, then the global interest list was not searched when distributing that event. In X11/NeWS, the pre-child interests of the ancestors of the canvas specified in a sent event are always searched. Window crossing and focus events are also not backwards compatible. Other than this, the changes are upwards compatible.

Summary

The X11/NeWS server design takes advantage of the great degree of commonality between the requirements of the X11 protocol and the requirements of the NeWS protocol. In X11/NeWS, one scheduler allocates time between clients of both protocols. One homogeneous window forest allocates screen resources between windows created by both protocols. One homogeneous event queue provides orderly distribution of events between clients of both protocols. One window manager is provided that presents an integrated user interface for manipulating all windows created by clients of both protocols. One keyboard focus manager is provided that presents an integrated user interface for directing keystrokes among clients of both protocols. One selection service provides exchange of data between clients of both protocols. Programs written to use either X11 protocol or NeWS protocol run unmodified, coexisting in an environment that presents an integrated interface to the user.

Acknowledgments

Credit is due to Jerry Farrell, Stuart Marks, David Rosenthal, and Sydney Springer for their contribution to the original design of the X11/NeWS server. Other members of the X11/NeWS development team, which include Nola Donato, John Erskine, James Gosling, Bob Rocchetti, and Tom Wood, have made major contributions to the design of areas of the server not covered in this paper. Acknowledgment is also due to David Lemke, Bruce Martin, and Helen Wong for developing large portions of the implementation, and especially to Steve Evans for managing the project. Warren Teitelman and S Page served as major reviewers of this paper.

References

- [1] Robert W. Scheifler.
X Window System Protocol Specification, Version 11.
Massachusetts Institute of Technology, Cambridge, Massachusetts and Digital Equipment Corporation, Maynard, Massachusetts, 1987.
- [2] Robert W. Scheifler, J. Gettys.
The X Window System.
ACM Transactions on Graphics, Vol. 5, No. 2, April 1986, Pp. 79-109.
- [3] Burns Fisher.
X11 Server Extensions Engineering Specification.
Digital Equipment Corporation, August 1987.
- [4] Sun Microsystems, Inc.
NeWS 1.1 Manual.
Sun Microsystems, Inc., PN 800-2146-10, 1987.
- [5] Sun Microsystems, Inc.
NeWS Technical Overview.
Sun Microsystems, Inc., PN 800-1498-05, 1987.
- [6] Adobe Systems, Inc.
PostScript Language Reference Manual.
Adobe Systems, Inc., Addison-Wesley Publishing Company, Inc., 1985.

Pseudo Devices: User-Level Extensions to the Sprite File System

Brent B. Welch
John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

A pseudo-device is a mechanism in the Sprite network file system that lets a user-level server process emulate a file or I/O device. Pseudo-devices are accessed like regular files or devices, and they exist in the file system name space. Pseudo-devices are implemented by transparently mapping client operations on the pseudo-device into a request-response exchange with a server process. The interface to pseudo-devices is general enough to be a transport mechanism for a user-level RPC system. It also provides a stream-oriented interface with write-behind and read-ahead for an asynchronous connection between clients and server. Sprite uses pseudo-devices to implement at user level its terminal drivers, the internet protocol suite, and the X-11 window system server. The pseudo-device implementation provides as fast or faster communication, both local and remote, than a UNIX UDP socket connection. †

1. Introduction

This paper describes pseudo-devices, a mechanism used to integrate user-level server processes into the Sprite operating system's distributed file system [Ousterhout88]. The distinguishing feature of pseudo-devices is that they appear in the file system name space and behave like regular files or devices. However, the operations on a pseudo-device are actually forwarded to a user-level process, called the server, which can implement them in any way it chooses. The motivation for pseudo-devices is to be able to move services traditionally found in the operating system kernel out to user-level processes. This keeps the size of the kernel down and it makes services easier to develop and debug. Sprite uses pseudo-devices to implement several system services including terminal drivers, the X windows server [Scheifler86], and the TCP/IP protocol family [IP81].

† This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

There are two advantages to using pseudo-devices to access user-level services: communication is via a traditional interface, and naming is via the existing file system name space. Pseudo-devices are accessed with the same `open()`, `read()`, `write()`, `ioctl()`, `select()`, and `close()` procedures used to access regular files and I/O devices. `Read()` and `write()` provide a byte stream interface which can be fully asynchronous with write-behind and read-ahead. `Iocontrol()` is synchronous and can be used as a transport mechanism for a Remote Procedure Call (RPC) system.

Using pseudo-devices, there is no need to invent a new name space to support user-level services. Names for pseudo-devices are protected, manipulated, and browsed in the same way as for other kinds of files. The Sprite file system provides name transparency across the network, so a pseudo-device can be accessed from any host in the network.

The remainder of the paper is organized as follows. Section 2 gives some background on the Sprite network operating system. Section 3 describes the pseudo-device interface from a client's point of view, and includes examples of services implemented as pseudo-devices. Section 4 describes the server's view of pseudo-devices and the implementation of the interface. Section 5 gives some performance measurements, including comparisons with other systems. Section 6 describes related work. Section 7 gives a summary and our conclusions.

2. Sprite Background

Sprite is a network operating system that we have developed from scratch at U.C. Berkeley over the last few years. Its design has been influenced by three things: multiprocessors, local area networks, and large physical memories. To support execution on a multiprocessor the Sprite kernel is multi-threaded internally, and Sprite supports shared memory between cooperating user processes. To support networks, Sprite uses a custom kernel-to-kernel RPC system [Welch86a] and a shared network file system that provides location transparent access [Welch86b] to files and I/O devices around the network. To exploit large physical memories, a distributed file caching scheme is used for high performance file access in an environment of diskless workstations [Nelson88]. The Sprite user interface is much like UNIX†, and for the purposes of this paper one can think of Sprite as a UNIX with a transparent network file system. Pseudo-devices could easily be implemented in a traditional UNIX kernel, and Section 6 describes some similar mechanisms that have been added to UNIX kernels.

3. Pseudo-device Clients

3.1. The Client Interface

Regular processes, called “clients”, access a pseudo-device the same way they access regular files and I/O devices. `Open()` names the pseudo-device and sets up a stream to it. `Close()` releases the stream. `Write()` and `read()` transfer data to and from the pseudo-device. `Select()` is used to wait until the pseudo-device is ready for I/O. `Iocon-`

UNIX is a trademark of AT&T.

`tr0l()`[†] is used for operations particular to the pseudo-device. These operations are identical, both in syntax and semantics, to the operations used for other files, so the implementation of the pseudo-device is transparent to the client.

3.2. Examples of Pseudo-Device Clients

Pseudo-devices are currently used for three purposes in Sprite: terminal drivers, network protocols, and window server communication. For example, for each terminal there is a pseudo-device and corresponding server process. Client processes make `read()` and `write()` requests on the pseudo-device instead of the terminal's serial line. The server implements the client's requests by manipulating the terminal's serial line in raw mode, and provides the full suite of 4.3 BSD `ioctl()` calls and line-editing functions such as backspace and word erase. The terminal driver is built as a library package so the same code is also used for managing `rlogin` streams and terminal-emulator windows. In this case, the pseudo-device mechanism provides a generalization of the 4.3BSD pseudo-tty facility.

The second use of pseudo-devices is to implement network protocols at user-level, TCP/IP in particular. While Sprite uses a custom network RPC protocol for kernel-to-kernel communication [Welch86a], the TCP/IP protocols are used to interface to non-Sprite systems, i.e. for mail service, remote logins, and remote file transfer. Client processes read and write a pseudo-device to use TCP, and the user-level server process implements the full protocol by reading and writing packets over the raw ethernet. In this case the internet pseudo-device provides the transport mechanism for a remote-procedure-call-like facility. A library package used by clients emulates the socket operations by issuing `iocontrols` on the internet pseudo-device. The internet server defines `iocontrol` operations for socket calls like `bind()`, `listen()`, `connect()` and `accept()`. The `iocontrol` input buffer is used to pass arguments from the client to the server, where the

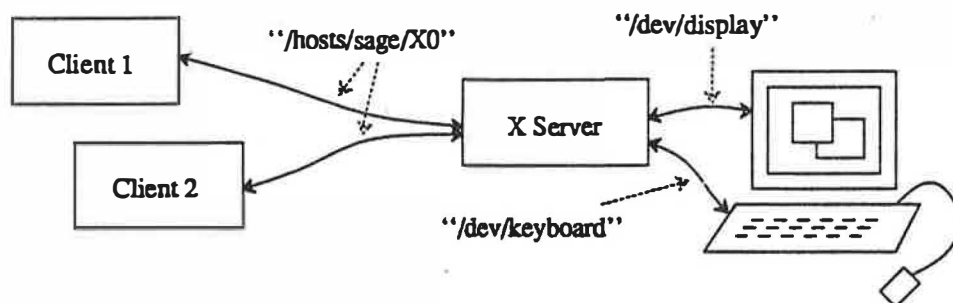


Figure 1. The pseudo-device `"/hosts/sage/X0"` is used by clients of the window system to access the X window server on workstation `"sage"`. The server, in turn, has access to the display and keyboard.

[†] `Iocontrol()` is a super-set of the standard UNIX `ioctl()` system call. `Ioctl()` takes only a command ID and a buffer as arguments. `Iocontrol()` takes a command ID, and 4 arguments that specify the size, and location, of an input and output buffers.

socket procedure is executed. The output buffer is then used to return results back to the client. Each library procedure in the client is simply a stub that copies arguments and results into and out of buffers and invokes the `iocontrol`.

The third use of pseudo-devices is to implement the X window server at user-level. For each display in the network there is a pseudo-device and a server process. Access to remote displays is not a special case because the file system provides network transparency. The X server controls the display and multiplexes the mouse and keyboard among clients, as shown in Figure 1. The clients use `write()` to issue commands to the X server, and `read()` to get mouse and keyboard input. A buffering system, which is described in detail in Section 4.2, provides an asynchronous interface between the window server and its clients to reduce context switching overhead.

4. Pseudo-device Server Implementation

The server for a pseudo-device is much like the server for any RPC system: it waits for a request, does a computation, and returns an answer. In this case it is the Sprite kernel that is making requests on behalf of a client process. The kernel takes care of bundling up the client's parameters, communicating with the server, and unpackaging the server's answer so that the mechanism is transparent to the client. The following sub-sections describe the implementation of this in more detail, including the I/O streams used by the server, the request-response protocol between the kernel and the server, and a buffering system used to improve performance.

4.1. The Server's Interface

The server for a pseudo-device is established when it opens the pseudo-device with the `PDEV_SERVER` flag. This open returns a *control stream* to the server. The server listens on the control stream for messages issued by the kernel each time a client opens the pseudo-device. These messages identify a new *request stream* the server gets for use in communicating with the client. Thus the server process has one control stream used to wait for new clients, and one request stream for each `open()` by a client process. These are shown in Figure 2.

The server accepts or rejects the client's open attempt when it handles the first request (an open) on a request stream. (Details of handling requests are given in the next sub-section.) If a client process subsequently forks (creates a new process), the new process shares the stream to the pseudo-device, and the server is not contacted. This is much cheaper than creating a new request stream each time a client process forks, and it maintains the UNIX semantics of shared streams. To the server, however, this means that there may be more than one client process using a request stream.

4.2. Request-Response

The Sprite kernel communicates with the server using a request-response protocol. The synchronous version of the protocol is described first, and then extensions to allow asynchronous communication are described. For each kernel call made by a client, the kernel issues a request message to the server and blocks the client process waiting for a reply message. Each request message includes the operation (open, close, read, write, `iocontrol`) and its associated parameters, which may include a block of data. Table 1 describes the contents of the request messages. The server replies to requests by making

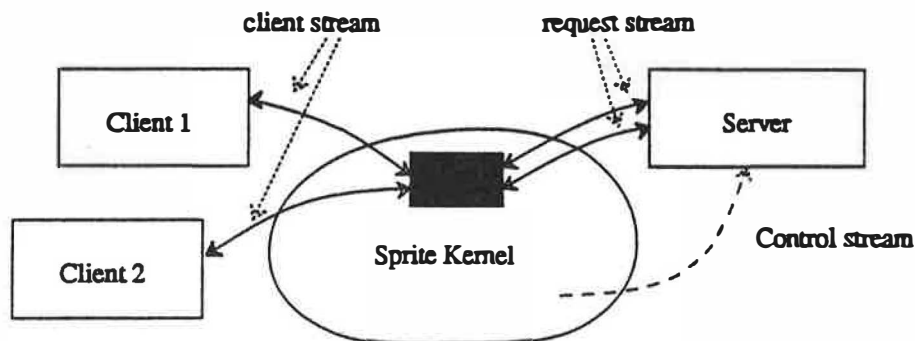


Figure 2. The control stream originates from the kernel and is used to pass new request streams to the server process. The client's streams are connected to corresponding request streams by a request-response protocol, which is represented by the black box in the figure.

an `ioctl()` call on the request stream. The `ioctl` specifies the return code for the client's system call, and the size and location of any return data for the call (i.e. data being read). The kernel copies the reply data directly from the server's buffer to the client's.

The kernel passes request messages to the server using a *request buffer*, which is in the server process's address space, and an associated pair of pointers, *firstByte* and *lastByte*, that are kept in the kernel and indicate valid regions of the buffer. With this buffering scheme the server does not read the request messages from its request stream.

Request Message Contents	
All Messages	Operation ID Process ID Input size Reply size
open	User ID (no data)
read	Byte offset (no data)
write	Byte offset Data block
ioctl	Command ID Data block
close	(nothing)

Table 1. All request messages have a standard header that indicates the operation (open, read, write, `ioctl`, close), the client's process ID, the the size of the input and result data. Additionally, each operation has its own specific parameters and, optionally, a variable size block of data.

Instead, the kernel puts request messages directly into the request buffer, and updates `lastByte` to reflect the addition of the messages. The server then reads a short message from the request stream that has the current values of `firstByte` and `lastByte`. The read returns only when there are new messages in the request buffer. After processing the message(s) found between `firstByte` and `lastByte` the server updates `firstByte` with an `iocontrol`. This buffering scheme permits the kernel to place several messages in the request buffer without waiting for each to be processed; the server can then process all of them at once, without requiring a context switch for each.

The buffering mechanism supports asynchronous writes (“write-behind”) at the server’s option. If the server specifies that write-behind is to be permitted for the stream, then the kernel will allow the client to proceed as soon as it has placed the write request in the server’s buffer. In enabling write-behind, the server guarantees that it is prepared to accept all data written to the stream; the kernel always returns a successful result to clients. The advantage of write-behind is that it allows the client to make several requests without the need for a context switch into and out of the server for each. On multiprocessors, write-behind permits concurrent execution between the client and server.

As a convenience to servers, the kernel does not wrap request messages around the end of the request buffer. If there is insufficient space at the end of the buffer for a new request, then the kernel blocks the requesting process until the server has processed all the requests in the buffer. Once the buffer is empty the kernel places the new request at the beginning of the buffer, so that it will not be split into two pieces. This is shown in Figure 3. No single request may be larger than the server’s buffer: oversize writes are split into multiple requests, and oversize `iocontrols` are rejected. If a write is split into several requests, the request stream is locked to preserve the atomicity of the original write.

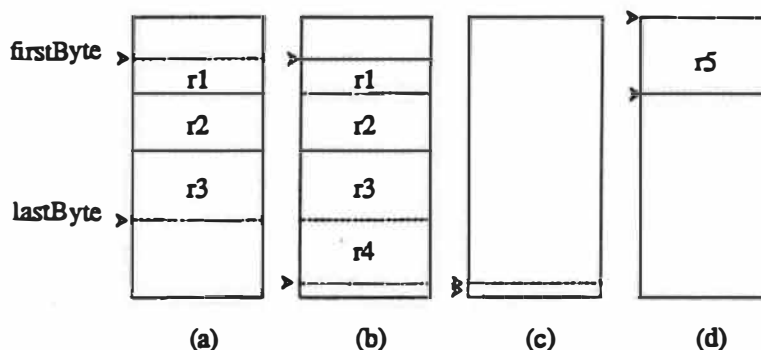


Figure 3. This figure shows the way the `firstByte` and `lastByte` pointers into the request buffer are used. Initially there are 3 outstanding requests in the buffer. The subsequent pictures show the addition of a new request, an empty buffer (the server has processed the requests), and finally the addition of a new request back at the beginning of the buffer.

Read performance can be optimized by using a *read-ahead buffer*. The server fills the read-ahead buffer, which is again in its own address space, and the kernel copies data out of it without having to switch out to the server process. Synchronization is done with `firstByte` and `lastByte` pointers as with the request buffer. In this case the server process updates `lastByte` after it adds data, and the kernel moves `firstByte` to reflect client reads.

To summarize the buffering scheme, the server has a request buffer associated with each request stream, and possibly a read-ahead buffer for each stream. These buffers are allocated by the server, and an `ioctl` call is used to tell the kernel the size and location of each buffer. The kernel puts request messages directly into the request buffer. The server's `read()` call on the request stream returns the current values of `firstByte` and `lastByte` for both buffers. The server updates the pointers (the request `firstByte` and read-ahead `lastByte`) by making an `ioctl()` on the request stream. The `ioctl` calls available to the server are summarized in Table 2.

4.3. Waiting for I/O

Normal I/O devices include a mechanism for blocking processes if the device isn't ready for input (because no data is present) or output (because the output buffer is full). To be fully general, pseudo-devices must also include a blocking mechanism, and the server must be able to specify whether or not the device is "ready". One possibility would be for the kernel to make a request of the server whenever it needs to know whether a pseudo-device is ready, such as during `read`, `write`, and `select`† calls. We initially implemented pseudo-devices this way. Unfortunately, it resulted in an enormous number of context switches into and out of server processes. The worst case was a client process issuing a `select` call on several pseudo-devices; most of the time most of the pseudo-devices were not ready, so the servers were invoked needlessly.

We subsequently re-implemented pseudo-devices so that the kernel maintains three bits of state information for each pseudo-device, corresponding to the readable, writable, and exception masks for the `select` kernel call. The pseudo-device server updates these

Server I/O Control Operations	
<code>IOC_PDEV_SET_BUFS</code>	Declare request and read-ahead buffers
<code>IOC_PDEV_WRITE_BEHIND</code>	Enable write-behind on the pseudo-device
<code>IOC_PDEV_SET_BUF_PTRS</code>	Set request <code>firstByte</code> and read-ahead <code>lastByte</code>
<code>IOC_PDEV_REPLY</code>	Give return code and the address of the results
<code>IOC_PDEV_READY</code>	Indicate the pseudo-device is ready for I/O

Table 2. The server uses these `ioctl()` calls to complete its half of the request-response protocol. The first two operations are invoked to set up the request buffer, and the remaining three are used when handling requests.

† The `select()` call is used to wait for several I/O streams at once. Each stream is identified by a small integer, and `select` takes as an argument a bitmask that has a bit set corresponding to each stream that is being waited on. `Select` is used to wait for streams to become readable, writable, or to have exceptional conditions.

bits with each reply ioctl and can also change them with the `IOC_PDEV_READY` ioctl. This mechanism allows the kernel to find out whether a pseudo-device is ready without contacting the server and resulted in a significant performance improvement for select. In addition, the server can return the `EWouldBlock` return code from a read or write request; the kernel will take care of blocking the process (unless it has requested non-blocking I/O) and will reawaken the process and retry its request when the pseudo-device becomes ready again. Thus the pseudo-device server determines whether or not the device is ready, but the kernel handles the logistics of blocking and unblocking processes.

4.4. Network Transparency

The Sprite file system provides a network-wide name space and remote device access so the pseudo-device server is not constrained to execute on the same host as its clients. An operation by a client on a remote pseudo-device is first shipped to the host running the pseudo-device server using the kernel's network RPC. The kernel RPC stub on the server's host then calls the regular pseudo-device routines to carry out the request-response exchange with the pseudo-device server. This is all transparent to both the client and the server processes.

4.5. Crash recovery

The client process is dependent on the server process to faithfully implement the pseudo-device. If the server hangs without returning a reply, then the client will also hang until the server process is killed or replies. If either the client or the server process dies, or their host crashes, then their stream to the pseudo-device automatically gets closed. When the client closes, the server gets a regular close request. If the server dies then current or future attempts by the client to use its stream will fail. Note that this applies even when write-behind is enabled.

5. Performance Review

This section of the paper presents a few performance measurements for pseudo-devices. There are a number of contributions to the communication cost: system call overhead, context switching, copy costs, network communication, synchronization, and other software overhead. For comparison, UNIX TCP and UDP sockets, and pipes under both Sprite and UNIX were also measured. The hardware used in the tests is a Sun 3/75 with 8 megabytes of main memory, and the UNIX is SunOS 3.2.

Each of the benchmarks uses some communication mechanism to switch back and forth between the client and the server process. Each communication exchange requires four kernel calls and two context switches. With pseudo-devices, the client makes one system call and gets blocked waiting for the server. The pseudo-device server makes three system calls to handle each request: one to read the request stream, one to make a reply, and one to update the `firstByte` pointer into the request buffer. With the other mechanisms the client makes two system calls: one to prod the server and another to wait for a response. The server makes two system calls as well: one to respond to the client, and one more to wait for the next request. The flow of control is shown in Figure 4.

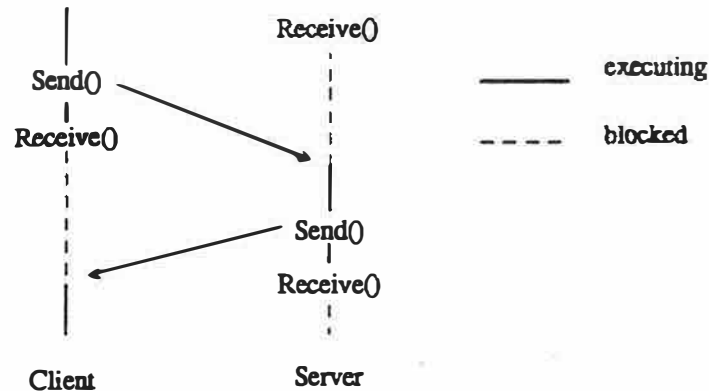


Figure 4. This shows the flow of control between two processes that exchange messages. Initially the server is waiting for a message from the client. The client sends the message and then blocks waiting for a reply. The client executes again after the server waits for the next request. Each benchmark has a similar structure, although different primitives are used for the Send() and Receive() operations shown here.

Process Communication Latency (microseconds)			
Benchmark	Bytes	Sprite	UNIX
PipeExchange	1	1910	2180
Pseudo-Device	0	2050	-
Pseudo-Device	1	2440	-
UDP socket	1	-	1940
TCP socket	100	-	5180
Remote Pdev	0	4260	-
Remote Pdev	1	5000	-
Remote UDP	1	-	4870
Remote TCP	100	-	7980

Table 3. The results of various benchmarks running on a Sun 3/75 workstation under Sprite and/or UNIX. Each benchmark involves two communicating processes: PipeExchange passes one byte between processes using pipes, Pseudo-Device does a null iocontrol() call on a pseudo-device, UDP exchanges 1 byte using a UNIX UDP datagram socket, and TCP exchanges 100 bytes (to avoid buffering in the protocol) using a UNIX TCP stream socket.

Table 3 presents the elapsed time for a round-trip between processes for each mechanism when sending little or no data. The measurements were made by timing the cost of several thousand round-trips and averaging the results. The measured time includes time spent in the user-level processes. In the second half of the table the client and server processes are on different hosts so there are network costs.

The difference between exchanging zero bytes and one byte using pseudo-devices highlights the memory mapping overhead incurred in the pseudo-device implementation.

When the kernel puts a request into the server's buffer it is running on behalf of the client process. On the Sun hardware only one user process's address space is visible at a time, so it is necessary to map the server's buffer into the kernel's address space before copying into it. Similarly, when the server returns reply data the client's buffer must be mapped in. The mapping is done twice each iteration because data is sent both directions, and obvious optimizations, i.e. caching the mappings, have not been implemented.

Figure 5 shows the performance of the various mechanisms as the amount of data varies. Data is transferred in both directions in the tests, and the slope of each line gives the per-byte handling cost. The graphs for UDP and TCP are non-linear due to the buffering scheme used in UNIX; chains of 112 byte buffers are used until a message is

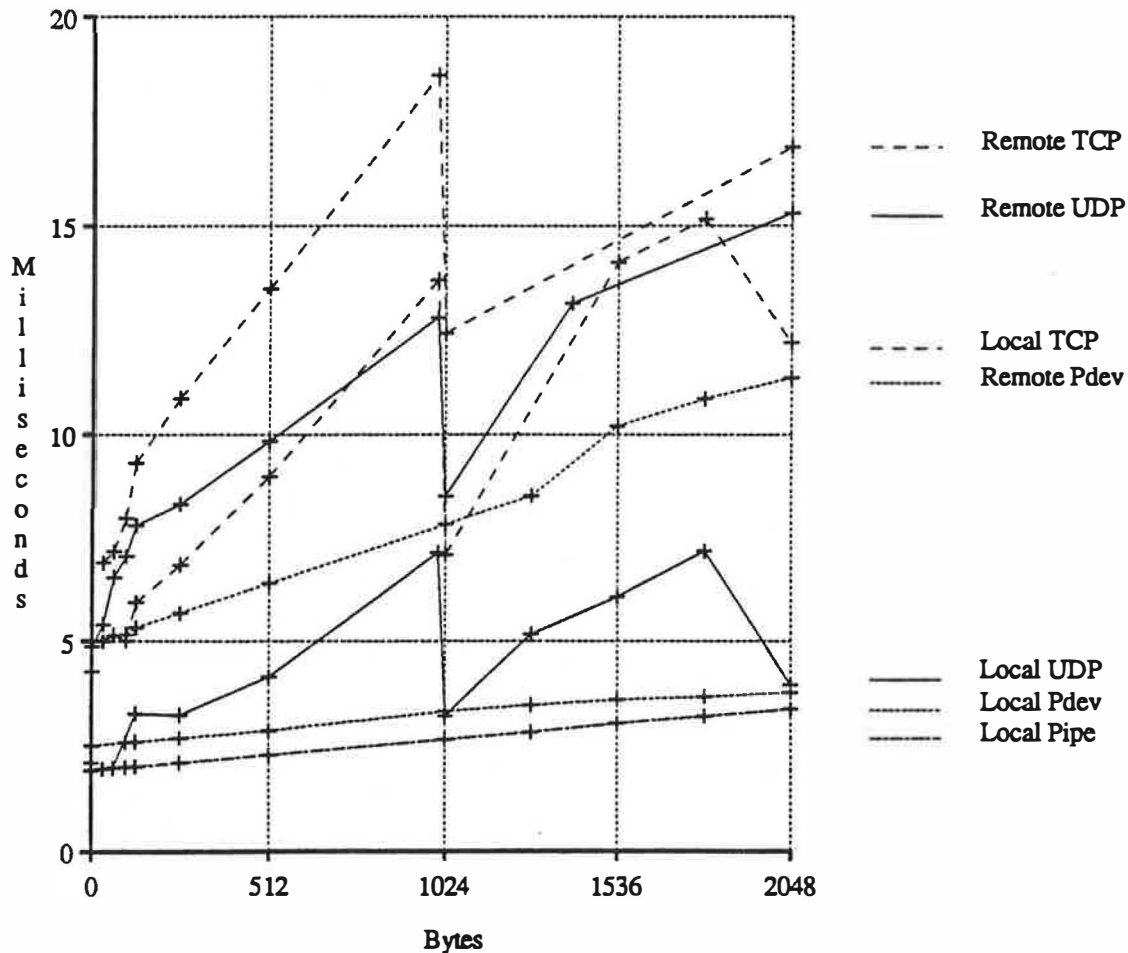


Figure 5. Elapsed time per exchange vs. bytes transferred, local and remote, with different communication mechanisms: Pseudo-devices, Sprite pipes (local only), and UNIX UDP and TCP sockets. The bytes were transferred in both directions. The shape of the UDP and TCP lines is due to the buffering scheme used in UNIX; chains of 112-byte buffers are used until a message is 1024 bytes, at which point chains of 1024-byte buffers are used.

1024 bytes, at which point chains of 1024 byte buffers are used.

The Sprite mechanisms have nearly constant per-byte costs. The unrolled byte copy routine used by the kernel takes about 200 microseconds per kbyte. Data is copied four times using pipes (there is an intermediate kernel buffer) and the measured cost is about 800 microseconds per kbyte. Data is copied twice using pseudo-devices, and we expect a per-kbyte cost of 400 microseconds. This is obtained when transferring between 1-kbyte and 2-kbytes, but we are not sure of the reason for the slightly higher cost at smaller transfer sizes.

In the remote case, the pseudo-device implementation uses one kernel-to-kernel RPC to forward the client's operation to the server's host. This adds about 2.2 msec to the base cost when no data is transferred, and about 4.3 msec when 1 kbyte is transferred in both directions. There is a jump in the remote pseudo-device line in Figure 5 between 1280 and 1536 bytes when an additional ethernet packet is needed to send the data.

The effects of write-behind buffering can be seen by comparing the costs of writing a pseudo-device with and without write-behind. The results in Table 4 show a 60% reduction in elapsed times for small writes. This speed-up is due to fewer context switches between the processes, and because the server makes one less system call per iteration because it doesn't return an explicit reply. The table also gives the optimal number of context switches possible, which is a function of the size of each request, and the actual number of context switches taken during 1000 iterations. Preemptive scheduling causes extra context switches. The server has a 2048-byte request buffer and there is a 40-byte header on requests, so, for example, 28 write messages each with 32 bytes of data will fit into the request buffer, but only one write message with 1024 bytes of data will fit. A scheduling anomaly also shows up at 1024 bytes; the scheduler preempts the client too soon so there are twice as many context switches as expected.

Pseudo-Device Write vs. Write-behind				
(Bytes vs. Microseconds & Context Switches)				
Size	Write	Write-Behind	Ctx Swtch	Opt Swtch
32	2330	910	40	36
64	2370	940	63	53
128	2400	1000	100	84
256	2450	1120	178	167
512	2590	1420	382	334
1024	3030	2660	2000	1000

Table 4. The elapsed time in microseconds for a write call with and without write-behind, and the number of context switches taken during 1000 iterations of the write-behind run vs. the optimal number of switches. The write-behind times reflect a smaller number of context switches because of write-behind. The optimal number of switches is not obtained because the scheduler preempts the client before it completely fills the request buffer. At 1024 bytes only one request fits into the server's request buffer, but there are extra context switches due to a bug in the scheduler.

6. Related Work

There are a number of features in existing operating systems that provide similar functionality to pseudo-devices, or that can be used to build up similar functionality. They fall into three categories, byte-stream mechanisms, message-based or RPC systems, and device-like mechanisms.

Message systems, including RPC implementations, are useful for implementing services outside the operating system kernel. However, they are usually not integrated into the file system name space, so an extra name service is required for connecting servers and clients. One good approach to this kind of system is found in the V-system. It has a uniform I/O interface that can be used to connect processes [Cheriton87]. To properly integrate itself into the V distributed name space, however, each server must handle naming operations as well. The pseudo-device interface is simpler in this respect as the kernel takes care of naming.

Byte-stream mechanisms, such as UNIX pipes, UNIX System V named pipes [Dolotta80], UNIX pseudo-terminals, and 4.3 BSD sockets, are limited to providing a reliable byte stream between processes. Usually any extra processing, like terminal line editing or a network protocol, is implemented inside the kernel. Pseudo-devices move the processing out of the kernel, and allow for more general operations via `ioctl()`.

The Version Eight stream facility [Ritchie84] also provides byte stream connections between processes, but it can be extended to allow emulation of an I/O device by a user-level process [Presotto85]. One extension converts `ioctl()` calls into special messages that appear in the byte stream to the server process. Another extension lets the server "mount" a stream on a file and return new streams to clients that open the file. This achieves the name transparency that pseudo-devices have, and lets the server multiplex itself among several clients. The main difference between pseudo-devices and the stream facility (aside from the underlying implementation) is the way the interfaces are used; the stream facility is designed to support different combinations of kernel-resident processing units, whereas the pseudo-device mechanism is oriented solely towards user-level implementation of services.

The watchdog facility proposed by Bershad and Pinkerton [Bershad88] provides a different way to extend the UNIX file system, but it can be used to achieve nearly the same effect as pseudo-devices. A "watchdog" process can attach itself to a file or directory and take over some, or all, of the operations on the file. The watchdog process is an un-privileged user process, but the interface is implemented in the kernel so the watchdog's existence is transparent. Watchdogs may either wait around for guarded files to be opened, or they are created dynamically at open-time by a master watchdog process. The main differences between the systems are that pseudo-devices provide the server with an asynchronous read-write interface to reduce overhead, and the watchdog process can handle subsets of file operations.

7. Conclusion

Pseudo-devices provide a way to integrate a user-level server process into Sprite's distributed file system. By making the service appear as a special I/O device, the existing open-close-read-write interface is retained. Pseudo-devices are named and protected just like other files. Byte stream communication is via `read()` and `write()`, and read-ahead and

write-behind can be used for asynchronous communication. Iocontrol() is available for operations specific to the pseudo-device, and can be used as the transport mechanism for an RPC system. The standard interface also means that the implementation of any specific pseudo-device could be moved into the kernel for better performance, or even implemented in hardware, without having to change any clients.

The performance of the implementation is acceptable at this point, although some further tuning may be possible. The buffering system represents our initial attempt to improve the performance and robustness of the system over an earlier pipe-based implementation. There is additional mapping overhead associated with copying data from one user process to another, so that pseudo-devices are not quite as fast as regular Sprite pipes. However, the request buffer is relatively simple for the server to manage, and the fact that it is pre-allocated allows some optimizations in the server.

We currently use pseudo-devices to implement terminal drivers, the X window server, and the TCP/IP protocol family. Future work includes extending the pseudo-device mechanism to a "pseudo-file system" mechanism that can be used to transparently access foreign file systems.

References

- Bershad88. B. N. Bershad and C. B. Pinkerton, "Watchdogs - Extending the UNIX File System", *USENIX Association 1988 Winter Conference Proceedings*, Feb. 1988, 267-275.
- Cheriton87. D. R. Cheriton, "UIO: A uniform I/O interface for distributed systems", *ACM Trans. on Computer Systems* 5, 1 (Feb. 1987), 12-46.
- Dolotta80. T. A. Dolotta, S. B. Olsson and A. G. Petrucelli, *Unix User's Manual, Release 3.0*, Bell Laboratories, Murray Hill, NJ, June 1980.
- Nelson88. M. Nelson, B. Welch and J. Ousterhout, "Caching in the Sprite Network File System", *Trans. Computer Systems* 6, 1 (Feb. 1988), 134-154, University of California, Berkeley.
- Ousterhout88. J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson and B. Welch, "The Sprite Network Operating System", *IEEE Computer* 21, 2 (Feb. 1988), 23-36.
- IP81. J. Postel, "Internet Protocol", *RFC 791*, Sep. 1981.
- Presotto85. D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Eighth Edition Unix System", *USENIX Association 1985 Summer Conference Proceedings*, June 1985, 309-316.
- Ritchie84. D. Ritchie, "A Stream Input-Output System", *The Bell System Technical Journal* 63, 8 Part 2 (Oct. 1984), 1897-1910.
- Scheifler86. R. W. Scheifler and J. Gettys, "The X Window System", *ACM Trans. on Graphics* 5, 2 (Apr. 1986), 79-109.
- Welch86a. B. B. Welch, "The Sprite Remote Procedure Call System", Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.
- Welch86b. B. B. Welch and J. K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", *Proc. of the 6th ICDCS*, May 1986, 184-189.

A UNIX[†] File System for a Write-Once Optical Disk.

Terry Laskodi

Bob Eifrig

Jason Gait

Computer Research Laboratory
Tektronix Laboratories
Tektronix, Inc.
PO Box 500 M/S 50-662
Beaverton, Oregon
97077

ABSTRACT

As hardware for write-once optical disks approaches magnetic disks in terms of performance, having a file system on an optical disk offers many capabilities not easily possible with a magnetic disk. This paper describes the design of an optical disk driver that can transparently support a standard UNIX file system. The implementation does not require auxiliary magnetic media, is transparent to applications, and requires minimal system changes.

1. Introduction

As write-once optical hardware approaches magnetic disk hardware in terms of performance, it is highly desirable to have the software to exploit some of the capabilities of optical disks[1]. Some of these capabilities are:

- Higher media densities, giving optical disks a better price/megabyte over magnetic disks.
- Stable medium. Optical disks are inherently more reliable than magnetic disks.
- Permanent backup. The medium is non-erasable.
- File versions. Because of the permanent nature of an optical disk, it is possible to provide file versioning, or audit trail, capability more easily than with magnetic disks.
- Removable media. Write-once optical disks are removable, making it easy to transport data between machines.

A file system provided by the host operating system has been the natural way of providing access to disk hardware, so it makes sense to use a file system for access to an optical disk; however, most file systems in use today share one implicit assumption: the medium on which the file system resides can be written multiple times on a per-block basis. This assumption is not true for write-once optical disks; therefore, current file systems cannot be used "as-is" with optical disk hardware. It would be impractical to invent yet another file system solely for the use of optical hardware. It is very desirable to make use of existing file system implementations, and as much as

[†] UNIX is a Trademark of Bell Laboratories.

possible, treat the write-once optical hardware as one would treat magnetic hardware.

2. Methods of Storing Data On A Write-Once Optical Disk

Many methods of storing data on a write-once optical disk have been suggested; the methods include an append-at-end of medium only[2], a copy-everything method[3], a hybrid method that stores data on the write-once disk and index information on a magnetic disk[4], and the use of B-trees[5] to store both data and index information on the write-once disk[6].

However, all of these methods have drawbacks, and the methods do make some implicit assumptions that may not hold true in the general case. The append-at-end of medium method is really only suitable for storing data in files that grow from the end, e.g., log files. The copy-everything method makes the implicit assumption that files are rarely updated, and when a file is updated, the entire file is copied to a new part of the disk. The hybrid method suffers from the problem of separating the index information from the actual data; any loss of index information usually means a loss of data accessibility as well. The B-tree method makes some implicit assumptions about the data that is being stored; specifically, it assumes that the data can be keyed upon to retrieve it. This is fine for standard data base applications, but it is not well suited for the more general case of file system usage.

The authors have implemented an alternative; it is more general than the above methods in that it makes no implicit assumptions about the data being stored. The method is more hardware oriented, based on a block replacement strategy, which makes it general enough to use in many different applications.

3. The Optical File Cabinet

In the UNIX file system, the interface between the higher level file system operations and the low level hardware is very well defined, consisting of a transfer direction, a logical disk block number where the transfer starts, and a transfer count.

With such a well-defined interface, it is possible to completely hide the fact that the hardware medium is write-once from the higher level file system operations.

In *The Optical File Cabinet*(OFC)[7], the relationship between the logical data and the physical storage is made indirect; this is done by translating the logical disk block number into a physical *optical* block number.

3.1 The File System Tree

In the OFC, the translation of logical block to physical block number is accomplished by the use of a *File System Tree*(FST), Figure 1, whose leaves contain the *physical* optical block corresponding to the logical disk block the operating system uses.

Each level of the tree either contains blocks of pointers to another level of the tree (called *interior nodes* of the tree), or they contain the *physical* optical block number that corresponds to the logical block number the operating system uses (or zero if that logical block has never been written to).

Assuming a one kilobyte optical block size, a two level tree (a tree where all of the entries point to a *physical* optical block) can support a 256 kilobyte file system, and a three level tree can support a 64 megabyte file system. In a three level FST, the root block contains pointers to the second level of the FST, and these blocks themselves contain the *physical optical* block corresponding to the logical block. One can also have a 16 gigabyte file system with a four level tree, with the second level of the FST pointing to another level of *interior nodes*.

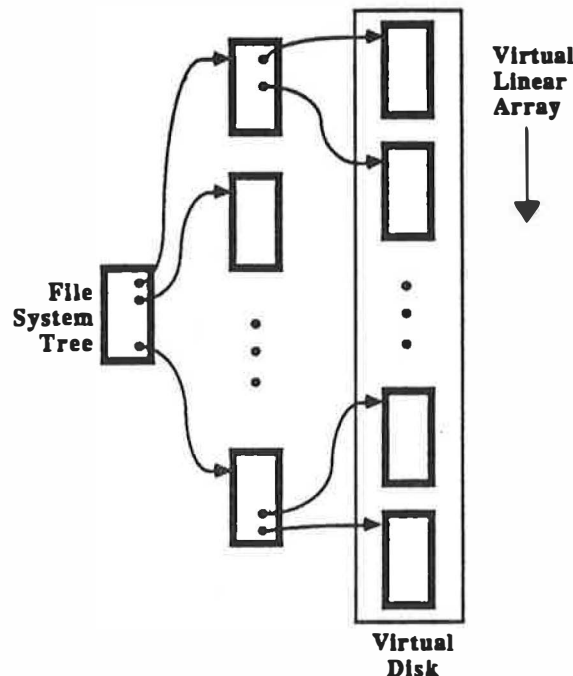


Figure 1. The File System Tree (FST). Note that the leaves of the tree are the physical optical blocks that correspond to the logical block the operating system sees. The pictured arrangement supports a 64 megabyte logical disk. Larger logical disks can be supported by adding additional layers of interior nodes to the tree.

By using efficient mask and shift operations, it is relatively easy to compute the respective offsets into an *interior Node*. For a 64 megabyte file system size, the computations needed are:

Offset into root node = $\text{block} \gg 8$

Offset into interior node = $\text{block} \& 255$

where \gg is the C right shift operator, and $\&$ is the C mask operator.

4. Data Structures For The OFC

Unlike the hybrid method, the OFC stores *all* data structures it needs on the write-once media. Thus, the OFC is self-contained and needs no magnetic disk to operate.

4.1 The Timestamp Structure

The *Timestamp Structure* is the center of all data structures used in the OFC, and contains all of the important information about the file system:

- **Pointers to other timestamps.** *Timestamp Structures* are doubly-linked together to form a timestamp hierarchy. There is a forward and backward pointer to other timestamps in the form of a *physical* optical block number.
- **Size information.** This includes the *logical* file system size, and the *physical* size of the disk. It is possible to dynamically change the logical file system size.
- **Partition information.** This is the standard UNIX partition information many device drivers use, consisting of a block offset and the size of the partition. It is included so that partitions may be dynamically changed without recompiling the driver.

- Various threshold values. One way to use the OFC is to let the *logical* file system size be a small percentage of the *physical* size of the disk, say 5-10%. This will leave room on the disk for updates. The threshold values control how much of the disk must be used before the user is notified to copy the current file system to a fresh disk.
- Pointer to the root of FST. Putting a pointer to the root of the FST in the timestamp gives the versioning capability described earlier.
- Various statistics. The OFC keeps statistics on various aspects of the system, such as *interior node* cache hit rate, proximal versus distal seeks[8], and the UNIX buffer cache size.
- Other incidental information. This includes a volume sequence number, a 32 byte volume name and 32 byte medium name, the creation date of the volume, and the size of the *interior node* cache. A version format number is also included so that different implementations of the OFC can coexist together, perhaps with one version having more capabilities than another.

The *Timestamp Structure* records a snapshot of the file system. The structures are doubly-linked together to form a hierarchy which records the history of the file system on the disk. The last *Timestamp Structure* recorded is the current state of the file system.

Since the *Timestamp Structure*, together with the FST, records a snapshot of the file system, the disk is always in a consistent state. If the system should crash without recording a *Timestamp Structure*, it is always possible to go back to an earlier snapshot of the file system.

4.2 Interior Nodes

Ideally, one would like to have all *interior nodes* of the FST in core at all times; unfortunately, this would consume quite a bit of memory. For a 64 megabyte file system, the *interior nodes* would take 256 kilobytes of memory.

In order to get around the large memory requirement, the OFC sets up and manages its own cache for the interior nodes. The cache uses an LRU method for replacement.

It is important to note that the data structures used in the OFC are completely independent of the data structures of the underlying file system. The OFC has no knowledge of the underlying file system format or of the handling of the disk buffer cache; these tasks are left to the higher level file system operations.

4.3 Checkpointing

It is imperative that a correct timestamp is written to the physical disk at frequent intervals. Writing a timestamp to the disk is referred to as a *checkpoint*; the interval is referred to as the *checkpoint interval*.

During a *checkpoint*, the state of the FST is frozen and written onto the physical device. The order of a *checkpoint* proceeds as follows:

- 1). Disk blocks in the operating system buffer pool cache are written.
- 2). The interior nodes of the FST are written.
- 3). The root block of the FST is written.
- 4). The new timestamp structure is written.

Since the timestamp is written last, the consistent state of the file system is always maintained. Because the medium is write-once, blocks are preallocated for the next timestamp in the hierarchy at checkpoint time.

5. Implementation Notes

Currently, the OFC is running on a version of UNIX 4.2 BSD. The optical hardware used consists of two Sony WDD-2000 write-once optical drives with a SCSI controller, utilizing 8" one gigabyte optical disks with a one kilobyte block size.

The OFC was installed in UNIX as a device driver, with no other changes other than adding new device driver entry points¹.

5.1 Initial Development

Since we had only a limited supply of optical disks, we developed a prototype system by simulating an optical drive with a magnetic drive. This prototype allowed us to implement and debug a major portion of the OFC, most notably the initialization of the OFC and managing the *interior node* cache, and to a lesser extent, crash recovery.

5.1.1 Initialization

When a file system is first opened in the OFC, the OFC first initializes the hardware, and then starts chaining through the timestamp hierarchy to find the latest timestamp on the disk. After the latest timestamp has been found, some simple crash recovery is attempted (if needed). After that, memory for the interior node cache is allocated.

5.1.2 Crash Recovery

Although the optical medium itself is inherently stable, other hardware problems can occur that can corrupt the medium. Controller failures can occur, along with other hardware failures of the underlying system. Software failures in other parts of the system can also corrupt the medium.

As part of the initialization procedure the OFC makes some consistency checks. First, the blocks for the next timestamp (which were preallocated at the last checkpoint) are checked to see if they have been written². If any of the blocks have been written (but not with a valid timestamp), then this is an error and that file system is corrupted. If the blocks are blank, then the OFC checks to see if the blocks immediately following them are blank. If any written blocks are found, then the disk is searched until a blank block is found, and the OFC verifies that it isn't just a "hole"³.

We found out very early that we needed this capability to find the first blank block on the disk that wasn't a "hole" for the block allocation strategy to work correctly.

5.2 Managing the Interior Node Cache.

Choosing to manage the interior nodes as an LRU cache made the chore much easier than we expected. We were able to make use of much of the existing code used to manage the UNIX disk buffer cache.

¹ Actually, this is not quite true. There was one other modification done, and that was to add code in the reboot code to force a *checkpoint* before rebooting (or halting). No other code was added, though.

² The Sony hardware used has capabilities to see if a block has been written to, or if it is blank, within a 256 block range.

³ We define a "hole" as any number of blank blocks, surrounded on both sides by any number of written blocks.

5.2.1 Data Structures for Managing the Cache.

We were also able to make use of many of the existing data structures used in managing the UNIX buffer cache, with just minor additions for some temporary storage used to hold results of calculations. We used the *struct buf* structure to hold all of the information of the interior node when it is incore. We hash incore interior nodes exactly the way the UNIX buffer cache is hashed; we use the *struct bufhd* structure for the hash chain headers. Memory for the buffers, along with the associated memory for the incore interior nodes, is allocated at initialization time.

5.3 Allocation of Disks Blocks on the Physical Media

Blocks are allocated on the physical media strictly from the start of the disk until the disk is full. During initialization, the next free block to be allocated is set to the first block *after* the blocks for the next timestamp that were preallocated at the *last* checkpoint before system shut-down. Blocks are allocated as they are needed (except for the blocks needed for a timestamp; they are the *only* blocks that are preallocated).

We chose this method for its simplicity. We never have to save the state of the block allocator during a checkpoint, and during initialization it doesn't cost us any extra time to recompute the first block available for allocation.

5.4 An Example Disk Transfer

We will give a specific example of a disk transfer occurring in the OFC. In this example, we have assumed a one kilobyte optical block size, and a 64 megabyte file system size.

In the UNIX file system, all disk transfers go through a separate routine called a *strategy* routine that is specific for each device. The strategy routine has one argument passed to it, a pointer to a *struct buf* structure. The structure has many elements in it used to help manage the buffer cache; the elements that are important to us are:

- A logical disk block number.⁴
- A transfer count.
- A transfer direction.
- A memory address for the transfer.

Assume the following parameters for a read transfer (logical block number 508, and a transfer count of 6144 bytes). After doing the normal UNIX driver verification for valid transfers, the transfer proceeds as follows:

- 1). Get exclusive use of the root block of the FST.⁵ The offset in the root block is computed as $508 \gg 8$, or 1. Now look in the cache of interior nodes for the block number for the first entry in the root LRU replacement of another interior node and then read in optical block 87 before proceeding.
- 2). Get exclusive use of the interior node for block 87. We are now at the second level of the FST. Once we have exclusive use of the interior node, release the root block.

⁴ The UNIX file system code assumes a 512 byte physical block size. We have to divide the logical block number by two to get a logical block number for use in computing the various offsets into the FST.

⁵ In normal UNIX buffer cache manipulation, this consists of checking to see if the buffer is busy; if the buffer isn't busy, mark it busy and proceed. If the buffer is busy, mark the buffer as wanted and wait for it to become available for use.

- 3). Now get the *physical* optical block number from the interior node for block 87. The offset into the interior node is computed as $508 \& 255$, or 252. If the physical optical block number is 0 (indicating that the logical block has never been written to), then zero out a block size buffer; otherwise, read the block into the appropriate place in the buffer.
- 4). Increment the starting *logical* block number, increment the buffer address by the *physical* block size, and decrement the transfer count by the *physical* block size.
- 5). Repeat until an error is encountered⁶, or until the whole transfer is completed. For the example given, this procedure is repeated six times.

If the transfer is a write, then the above procedure is modified just slightly. The traversal of the FST is exactly as described above, but with the following modification: at the lowest level of the FST, new physical optical blocks are allocated to write the new data in the buffer on the disk. Next, because we are logically modifying an interior node of the tree, we have to allocate a new physical optical block for that node, plus all of its ancestors up to the root of the FST.

In practice, we do not allocate a new physical optical block every time we change an interior node; we only allocate it when we do the first logical write into the interior node, and then mark the interior node "dirty". Once an interior node is marked "dirty", all write transfers then proceed as a read transfer. When the interior node is written onto the disk during a checkpoint, we then mark the interior node "clean".

Note that because of using the UNIX buffer cache code to manage the interior nodes as an LRU cache, it is possible (and it is always true for multi-block transfers) to sleep() in the strategy routine waiting for an interior node to not be busy.

6. Optimizations.

In the last section, we gave an example of a disk transfer occurring in the OFC. Now we will discuss some of the implications of the method used. Since a user's impression of a file system depends mainly on the read response time⁷, we optimize for reads as much as possible.

First, it is important to note that all of the I/O in the OFC is accomplished one kilobyte at a time⁷, and each one kilobyte transfer requires a complete traversal of the FST. This slows down multi-block I/O requests, and it was most noticeable when we were prototyping with a magnetic disk.

The first optimization we made was to try and do all of the transfer with just one I/O request; in order to do this, we constrain a transfer to use just one interior node at the lowest level of the FST⁸; however, just constraining the transfer to use one interior node is not sufficient. If the transfer is a read⁹, we also have to make sure that all of the physical optical blocks recorded in the interior node are all physically consecutive on the media. If all of the physical blocks are consecutive, then we can do the I/O with just one request.

In practice, we do as much of the I/O as we can with one request, splitting the transfer into the largest pieces possible.

⁶ Such as a bad block, or user errors. User errors occur only in *physio* usage of the file system.

⁷ Or the physical optical block size.

⁸ We must do this in order to avoid the classic deadlock problem, locking a database in two different places, possibly by more than one party.

⁹ If the transfer is a write, then constraining the transfer to the same interior node is sufficient, since we have to allocate new physical optical blocks. The block allocator has a block count parameter passed to it.

Second, in the prototype version of the driver, all I/O was synchronous instead of the normal asynchronous behavior a UNIX system expects. We added some new state variables to the driver and, as much as possible, tried to keep the asynchronous nature.

Another optimization we made concerns reads; if the first block of the transfer has an optical block number of zero at the lowest level of the FST, then we keep track of how many optical block numbers are zero at the lowest level, and then zero out enough of the buffer with just one call to `bzero()`.

The last optimization concerns writes. For a write transfer, in the first version of the driver using the real optical hardware, we checked to see if the blocks being written to were actually blank using the blank sector search capabilities of the hardware. We did this to verify that there were no bugs in either the block allocator or the initialization code. Unfortunately, the blank sector search capability of the hardware is very slow, and this noticeably slowed write transfers. Now this check is made optional, with the default to not do the check.

7. Simulations and Measurements.

We did some simulations to convince ourselves that implementing a file system on write-once media was possible. We now discuss the simulations results versus the actual measured results.

One simulation was varying the size of the interior node cache and see how the cache hit rate varied. See Figure 2.

Even for small cache sizes, the measured hit rate is much higher than the simulated hit rate. The simulation assumed a very pessimistic view of file system behavior, specifically a very random behavior. The measured hit rate can be explained in terms of *locality of reference*, i.e. the measured behavior is more uniform and not as random.

7.1 Measurements.

In this section, we compare the time it takes to process a 1/2 megabyte file (both read and write times). We list the times for the prototype system using a magnetic disk, and the real system with the optical hardware itself, along with times from a "regular" file system on a magnetic disk. See Table 1.

The times for the prototype system are particularly interesting in that they are a very good measure of the software overhead in using the OFC. Note that all times listed are the *total* time reported by the time command, and the I/O requests were 4096 bytes.

As can be seen from the times listed, there is no significant software overhead in using the OFC; however, there is significant hardware overhead with the Sony optical hardware used.

The Sony disk drive used rotates with a constant linear velocity, varying from 900 RPM to 535 RPM; the average seek times are 300 msec, with a seek time of 20 msec for less than 50 tracks.

8. Other Device Driver Capabilities

The driver has some unique capabilities a user can exploit, and they are included mainly for ease of user programming, though the driver does not need to make use of them. Some of these capabilities include:

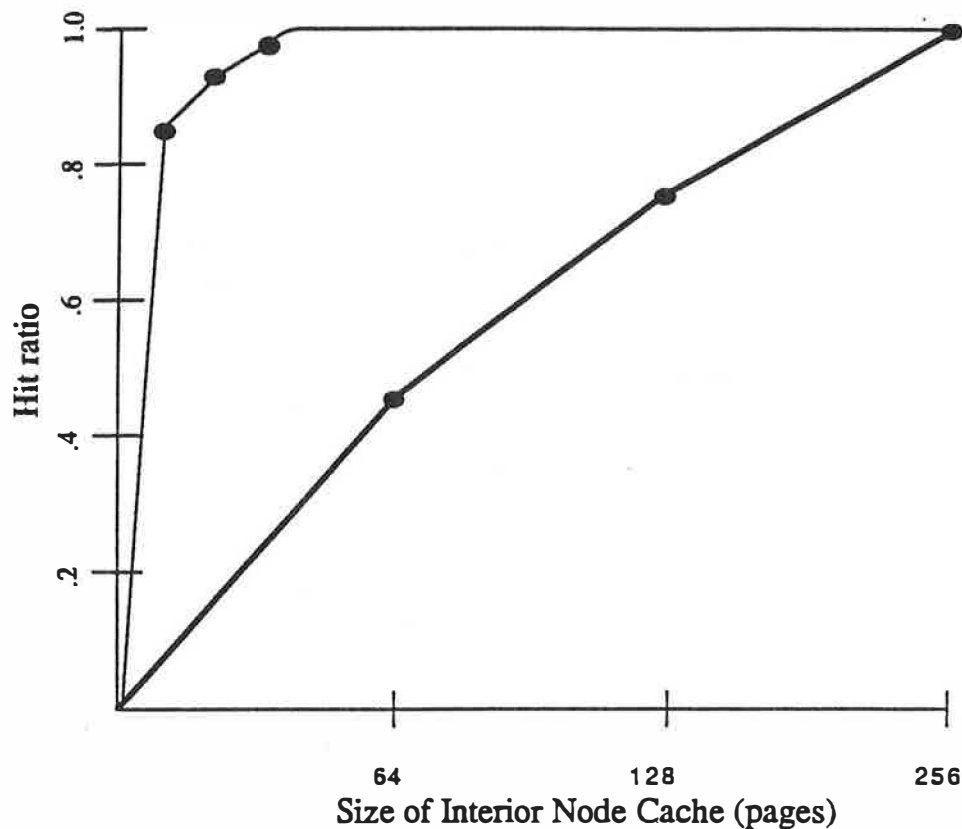


Figure 2. Size of interior node cache versus cache hit rate. The solid line is simulated results, the thin line measured results.

System	Read		Write	
	Single Block	Multi-Block	Single Block	Multi-Block
Regular	N/A	0:04.1	N/A	0:04.4
Magnetic	0:13.7	0:03.3	0:18.2	0:04.6
Optical	1:23.4	0:21.2	2:47.5	0:34.3

Table 1. Time to process a 1/2 megabyte file. Regular refers to normal file system usage, magnetic refers to the prototype system on a magnetic disk, and optical refers to the real system running on the real optical hardware.

- Translation of logical block number to physical block number, doing a complete traversal of the FST.
- For medium change, the capability to get exclusive use of a device and shut off all other file system usage.
- The capability of dynamically turning on/off the write-check verification.
- The capability to dynamically change the size of the interior node cache.
- The capability to bypass the normal initialization of the timestamp hierarchy and to initialize the file system with a specific timestamp.
- The capability to dynamically change the threshold values.

- The capability to get the current timestamp without having to read the disk.
- The capability to do a checkpoint.
- The capability to do an arbitrary SCSI command.

9. Conclusions

Our experience has shown that it is possible to implement a UNIX file system on a write-once medium, and to hide this fact from the higher level file system code. The prospect of having a file system on a media with high reliability and permanent nature is very exciting.

Although the OFC was installed as a device driver with almost no change to other operating system software, the authors can envision a system that knew about the OFC and can make use of some of its features. The operating system could provide an automatic version capability, with fast access to previous versions of files with the right data structures maintained on the write-once medium.

Taking further advantage of the permanent nature of optical disks, one could checkpoint the entire operating system, shut the system down, and bring it back up to the exact same environment that existed at checkpoint time[10]. The possibilities seem endless with the capabilities of optical disks.

References

- [1] Fujitani L., *Laser Optical Disk: The Coming Revolution In On-Line Storage*, CACM, 27(1984)546-554.
- [2] Finlayson R., Cheriton, D., *Log Files: An Extended File Service Exploiting Write-Once Storage*, Proceedings of the 11th Symposium on Operating Systems Principles, Nov. 1987, 139-148.
- [3] Garfinkel, S., *A File System For Write Once Media*, MIT Media Lab, Oct. 1986.
- [4] Schaffer, J.B., Schelin, J.W., Thomas, J.T., *Data Base and File Management Approach for Large Optical Disk Systems*, Proceedings SPIE 421, 1983, 20-30.
- [5] Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison Wesley, 2nd Edition, 1973.
- [6] Rathman, P., *Dynamic Data Structures on Optical Disks*, Proceedings IEEE Data Engineering Conference, April 1984, 175-180.
- [7] Gait, J., *The Optical File Cabinet: A Random Access File System for Write Once Optical Disk*, IEEE Computer, May 1988.
- [8] Christodoulakis, S., Faloutsos, C., *Design and Performance Considerations for an Optical Disk Based Multimedia Object Server*, IEEE Computer, Dec. 1986, 45-56.
- [9] Vitter, J., *An Efficient I/O Interface for Optical Disks*, CS-84-15, Dept. of Computer Science, Brown University, 1984.
- [10] Gait, J., *A Checkpointing Page Store for Write Once Optical Disks*, IEEE Transactions on Computers, to appear.

The File Motel — An Incremental Backup System for Unix

Andrew Hume
research!andrew
andrew@research.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This report describes an incremental backup system that automatically tracks and stores modified files from a group of client Unix systems on a central backup system. Names of backed up files are stored in a database which can be interrogated easily by users. Backup copies are normally kept forever. The system requires very little manual intervention and works with any archival media but is oriented towards use with write-once optical disks.

1. Introduction

This report describes an incremental backup service for the Computing Science Research Center. The center has about 70 users spread over about 17 computers. The typical computer is a VAX 11/750, but there is a Sequent Balance 8000, several Sun III workstations, two VAX 8550s and a Cray X/MP-24. Most of the computers are connected by the Datakit local area network, some by Ethernet and a few by both. Access to files and programs on other machines comes in two flavours, special purpose commands that copy files or execute remotely, and the network file system[1], which makes files on other systems appear to be on your system almost transparently.

A backup system allows access to some number of versions of a given file. Historically, our center has employed two methods. On some machines, important file systems are backed up onto tape using *dump(8)* or *dd(1)*. Other, more relaxed, machines copy file systems onto spare file systems once a week or so. Some paranoid lazy users copy important files to another system which gets backed up properly. Due to these behaviours, between 40–60% of the center's disk capacity is devoted to backup copies of various flavours. Restoring files over a week old is generally tedious (if possible at all); manual processing is required to find (and then mount) the dump tape and then to restore the file.

The system described here is an incremental system running at user level. Rather than copying file systems as a whole, it scans the file system for files that have changed recently and saves only those files. It has the advantages that for the user, it is automatic and needs no overt action to activate backup. Because the versions of a file are recorded in a database, it is easy to search for a particular version of a file without having to pore through tapes or other media. The main disadvantage is that not all files are backed up (for example, corefiles and object files are not normally saved), and the particular file you want may not have been saved. However, a user may explicitly save any particular file.

While the backup system can use any kind of medium for archiving backup copies of files, the medium of choice is write-once-read-many (WORM) optical disks. These have a similar price per bit to magnetic tapes, last longer (approx 30 years), are much more compact, are easier to use, and for the typical operation of mounting a disk and recovering a single file, are significantly faster than tape.

The next two sections describe the underlying model for the backup system and a description of the user interface. The next section describes the grimy details that make the system work; it is best omitted by the casual reader. The final section describes the lessons learned during the two years of system development.

2. Architecture

The underlying model of the backup system is a group of *client systems* saving copies of selected files on a central *backup system*. (The backup system may be, and normally is, a client system as well.) The backup system stores backup copies as files with internally generated names. A database links the original filename and modification time to the backup copy name. The exact contents of the database are described below.

Files are transferred from the client to the backup system daily (the period is configurable per site). The transfer takes several steps. The client prepares a list of files to be backed up by executing the shell script `/usr/lib/backup/sel`. Although this script is site dependent, it normally includes any file whose pathname starts with `/usr`, has a size less than 512KB, has been modified in the last seven days, and is not a known form of "junk" (such as corefiles). The size limit is just for administrative convenience. Note that this list contains many files that should have already been backed up. This list is then sent to the backup system which returns the list of the files the backup system does not have a copy of. Our typical system sends about 300–400 names; the backup system returns 70–100. These files are then copied by the client to a receiving area on the backup system.

Periodically, the backup system processes the files in its receiving area, assigns backup copy names, makes the copies to a trusted area and then updates the database. The backup copy names are simple volume—file pairs (such as `v456/12`) described later. Periodically, files in the trusted area are dumped onto some archival medium.

The database supports two binary relationships. The most common is file version to backup copy. A file version is represented as a pathname catenated with the modification time (or more exactly, the `ctime`, see `stat(2)`). For example, the most recent copies of wild's password file are

```
/n/wild/etc/passwd//556840921 -> v935/206
/n/wild/etc/passwd//557609109 -> v978/123
/n/wild/etc/passwd//557774351 -> v987/484
/n/wild/etc/passwd//559328298 -> v1083/392
```

The second relationship relates files to their most recent backup copy. Continuing the above example,

```
/n/wild/etc/passwd -> 559328298
```

Thus, finding the most recent backup copy for a file requires two searches; one to get the time and the second to get the copy name.

What does this mean to the user? Recovering a file implies selecting the desired version (not always the most recent) and then accessing that particular copy. The most common difficulty is knowing a file's true name. For example, a file you can access as `/usr/gonzo/poot` may be stored with the name `/usr1/guest/gonzo/poot` (particularly, if `/usr/gonzo` is a symbolic link to `/usr1/guest/gonzo`). Generally, the client system administrator can fix this but then the system knows both names! (Hopefully, the older names can be forgotten over time.) The other major difficulty is that files may not get backed up the first (or second ...) night after they are modified, say because of some network or machine failure. Generally they are, but a cautious user would check if the backup exists (via the `backup grep` command) before depending on it. The main advantage for the user is a quick determination of what versions exist and where they are. It allows efficient answers to requests like "what files in this directory have been changed since *date*?" or "construct this directory as it was on *date*".

3. User interface

The user interface comes in two rather different styles. One is a set of tools which give full and efficient access to the backup system. Another is a simulated file system which sacrifices some functionality and performance in order to present the backed up files as ordinary files and directories.

3.1. The backup file system

The backup file system is a remote file system using the second version of the Weinberger file system protocol[1]. The command

```
backup mount xx
```

mounts a backup file system on *xx*. Backup files may then be accessed in the normal way. For example,

```
$ cd xx/n/wild/etc
$ diff passwd /etc/passwd
```

The backup file system has two peculiarities. The first is that any particular invocation has an associated date. Backup copies more recent than this date do not appear in the file system. The date is set (and read) by `backup date`. The second is that for every file or directory (say with name *fileordir*) there exists a directory *fileordir.V* containing all versions of *fileordir*. Note that *fileordir.V* exists only when explicitly mentioned:

```
$ cd xx/n/wild/etc
$ ls -l ttys*
-rw-r--r--  1 root  bin    1162 Feb 29 11:08 ttys
$ cd ttys.V
$ ls -l
-rw-r--r--  1 root  bin      876 Apr 22  1986 0
-rw-r--r--  1 root  bin      876 Jul 22  1986 1
.
.
.
-rw-r--r--  1 root  bin    1036 Oct  7  1987 14
-rw-r--r--  1 root  bin    1162 Dec 16  1987 15
-rw-r--r--  1 root  bin    1162 Feb 29 11:08 16
```

The properties of the *.V* directories ensure that they are easily accessible to the user but will not interfere with copying of directory trees.

The backup file system is unmounted by

```
$ backup unmount xx
```

If a backup file system is unused for more than an hour or so, it may silently disappear.

3.2. The backup tools

The two primary tools are *backup grep*, which examines the backup database, and *backup fetch*, which takes backup copies and restores them as files. The command *backup recover* provides a simpler interface to these commands that is suitable for most purposes.

Backup grep looks for files. (It has many options; some are described below.) The first, and simplest, is by exact name:

```
$ backup grep /n/wild/etc/passwd
/n/wild/etc/passwd//Tue Sep 22 12:58:18 1987
$ backup grep -d /n/wild/etc/passwd
/n/wild/etc/passwd//559328298
```

The `-d` prints the date as a number, rather than as a string. If the file does not exist in the backup database, nothing is printed. If a filename looks like the latter (`/n/wild/etc/passwd//559328298`), the name of the backup copy is printed instead:

```
$ backup grep /n/wild/etc/passwd//559328298
/n/wild/backup/v/v1083/392
```

This is how *backup recover* determines which copy (the most recent) to recover.

Another common action is to retrieve some older version of the file (often the second last copy, the one before you wrecked the file). The `-v` (for versions) option prints out all the versions of a file:

```
$ backup grep -v /n/wild/etc/passwd
/n/wild/etc/passwd//Sun May 25 03:18:49 1986 /n/wild/backup/v/v0/279
*
*
*
/n/wild/etc/passwd//Mon Aug 24 18:02:01 1987 /n/wild/backup/v/v935/206
/n/wild/etc/passwd//Wed Sep 2 15:25:09 1987 /n/wild/backup/v/v978/123
/n/wild/etc/passwd//Fri Sep 4 13:19:11 1987 /n/wild/backup/v/v987/484
/n/wild/etc/passwd//Tue Sep 22 12:58:18 1987 /n/wild/backup/v/v1083/392
```

Any version can be retrieved, although older versions may be kept on less convenient media. To recover the password file as of September 1, type

```
$ backup fetch /n/wild/backup/v/v935/206
```

To recover the most recent copy, simply say the filename:

```
$ backup recover /n/wild/etc/passwd
```

The original name of a file is stored inside the backup copy and this is the default destination of the backup copy. There are two ways to change the output name. The first, `-o dir`, places the file in the specified directory. The second, `-D old=new`, replaces an initial substring of the original name with a specified string. The following two commands yield the same result:

```
$ backup recover -o /tmp /n/wild/etc/passwd
$ backup recover -D /n/wild/etc=/tmp /n/wild/etc/passwd
```

The former form is intended for convenient recovery of a single file. The latter form (an example is shown below) is better suited to recovering file trees. Note that recovering a directory does not recover any of the files in it.

Sometimes, you may not be sure of the file name. The `-D prefix` (directory) option finds the most recent copy of all the filenames beginning with *prefix*. For example

```
$ backup grep -D /n/wild/etc
/n/wild/etc/crontab//Fri Oct 9 10:17:03 1987 /n/wild/backup/v/v1176/93
/n/wild/etc/fstab//Wed Sep 30 13:41:17 1987 /n/wild/backup/v/v1119/571
/n/wild/etc/group//Tue Jul 28 16:21:45 1987 /n/wild/backup/v/v754/594
/n/wild/etc/passwd//Tue Sep 22 12:58:18 1987 /n/wild/backup/v/v1083/392
/n/wild/etc/rc//Sun Oct 11 16:30:27 1987 /n/wild/backup/v/v1180/16
/n/wild/etc/ttya//Wed Oct 7 15:36:58 1987 /n/wild/backup/v/v1151/363
$ backup grep -D /n/wild/etc/c
/n/wild/etc/candest//Tue Nov 27 14:10:57 1984 /n/wild/backup/v/v0/285
/n/wild/etc/crontab//Fri Oct 9 10:17:03 1987 /n/wild/backup/v/v1176/93
```

As a last resort, you can look for regular expressions with the `-e` option. This currently takes several hours to complete on a VAX 11/750 (there are many filenames to search). As an alternative, we keep an almost up to date listing of backed up files in

/n/wild/usr/spool/ilenames. Although this file is large, *egrep*(1) is fast[2].

Regrettably, the *-D* option does not really recover the most recent state of a directory as member files deleted long ago will be cheerfully recovered. Although this difference rarely matters, the only correct way is to use the *-F* option to recover the directory as a file and use *ls* to list the extant member files. It is probably easier to use the backup file system described above.

Most of the above options for *backup recover* are simply passed through to the program that really does the work, *backup fetch*. (If your backup system uses an optical disk, the program is *backup wfetch*.) *Backup fetch* takes multiple arguments, either on the command line or on standard input. The arguments should be backup copy names. As a convenience, all characters up to and including the last blank in an argument are deleted. Thus, the output from *backup grep -D* can be used directly. For example, to recover the whole directory tree of files rooted at /n/wild/usr/andrew as a tree rooted at /tmp/andrew:

```
$ backup grep -D /n/wild/usr/andrew | backup wfetch -D /n/wild/usr=/tmp/
```

If you need reassurance, *backup fetch* will chatter at you if given the *-v* (verbose) option.

The normal recovery procedure looks like this (assuming you have WORMs):

```
$ backup recover -o . /n/wild/etc/passwd
looking on optical disk
starting worm server (it takes a couple of minutes)
/n/wild/backup/v/v1083/392 -> ./passwd
```

The server hangs around for a while so that if we ask for another file shortly afterwards, it goes much quicker:

```
$ backup recover -o . /n/wild/etc/crontab
looking on optical disk
/n/wild/backup/v/v1283/197 -> ./crontab
```

It may be that the file is not available. In that case, *recover* (actually *wfetch*) tells you which disk to mount:

```
$ backup recover -o . -m v935/206
looking on optical disk
need disk backup2a
```

If you have a WORM jukebox, it will mount any disks it needs itself. As far as the optical media are concerned, only the last two components of the backup copy name are significant. Thus

```
/n/wild/backup/v/v935/206
v935/206
```

denote the same backup copy.

Backup stats gives statistics about the number and average size of files saved over all users and all systems. The output can be restricted to certain users by *-uusera,userb,...,userx*, and/or certain systems by *-ssysa,sysb,...,sysx*. The user or system name *** matches all names. The *-i* option gives statistics about each individual user/system. For example

```
$ backup stats -uandrew,norman -sdutoit,wild
* * [105]: 7591(72.3) files, 101.636(0.968) MB; 13.4 KB
$ backup stats -uandrew,norman -sdutoit,wild -i
wild norman[102]: 60(0.6) files, 0.205(0.002) MB; 3.4 KB
wild andrew[105]: 2925(27.9) files, 49.556(0.472) MB; 16.9 KB
dutoit andrew[102]: 303(3.0) files, 5.022(0.049) MB; 16.6 KB
dutoit norman[102]: 4303(42.2) files, 46.853(0.459) MB; 10.9 KB
```

The number of days statistics are available for is shown inside []. The figures in () are averages per day. Trends can be revealed by the `-d` option; it reports averages for the last one, seven and thirty days:

```
$ backup stats -uandrew -d
                1d          7d          30d
* *:          275, 3.1MB    290, 3.3MB    206, 2.6MB
```

4. Behind the scenes

The client systems normally send files to the central backup system daily. After all the clients have finished, the backup system does some additional processing.

The client systems are bound to the backup system by the contents of the file `/etc/backup` which contains three lines. The first contains a (network file system) path-name for the backup database directory. The second contains a network address for the backup machine. Within our Center, the backup machine is called `wild` and its Datakit address is `nj/astro/wild`. The third indicates if you have optical disks; it is blank if you do not. Otherwise, it is the string `worm` or the name of the default WORM drive. Our `/etc/backup` looks like this

```
/n/wild/backup
nj/astro/wild
worm
```

The clients send the files to be backed up by executing the command

```
/usr/lib/backup/sel | /usr/lib/backup/act
```

Each client is free to customise `sel` but a rudimentary version might look like:

```
/usr/lib/backup/fcheck 512 7 /usr/* | \
sed -e '/\.\.o$/d
/\a\.out$/d
/\core$/d'
cat <<'EOF'
/etc/passwd
/etc/ttys
/etc/crontab
/etc/rc
EOF
```

`fcheck` is akin to `find`; takes a maximum size (in 1024 byte blocks), a minimum age (so many days before now) and a list of names. Symbolic links are followed only if explicitly mentioned; otherwise, they are saved as a symbolic link.

A simplified version of `act` is (eliding checksumming and networking detail)

```
me=`cat /etc/whoami`
w=/tmp/$me
m=`sed -n 2p /etc/backup`
/usr/lib/backup/iprint > $w.bi
rx $m /usr/lib/backup/notdone $me < $w.bi > $w.bo
/usr/lib/backup/bpush $me < $w.bo
```

(Berkeley systems would use `rsh` rather than `rx`.) The list of filenames has a time appended by `iprint` and is given to `notdone`, running on the backup system, which reads this list and outputs only the filenames that need be backed up. This new list is then fed to `bpush` which copies the corresponding files to receiving areas on the backup machine. In the process, the files become *backup copies* because of a 1KB header prepended to the file. The header contains


```

version number
the file's inode (struct stat)
the file's security label (struct label)
checksum
owner UID (as a string, say bin)
owner GID (as a string, say other)
original pathname
backup copy name

```

The security label is only used in an experimental system implementing a more secure Unix[3]. Note that all the information needed to build the database and to restore the file is stored in the header.

All the files in the holding area are owned by the special user *daemon*. Files are mode zero while they are being received. When the copy has completed successfully, the mode is set to 0600. The backup copy name is set to the empty string. The holding areas are directories each with 32 subdirectories, *rcv0* through *rcv31*. The names of the holding areas are listed in the file */backup/rcvdirs*, ours contains */backup/rcv*. When a holding area can't hold any more (presumably because the file system has run out of space), the next holding area is used. To protect against running out of space while writing a file, the file is initially written with zeros.

In the next stage, the backup system processes files in the holding areas and transfers them to trusted storage. It is preferable, but not critical, that all the clients have finished sending files; the tardy files will be processed the next day. This stage is executed by the script */usr/lib/backup/munge*, which executes the following code in each receiving directory:

```

ls | /usr/lib/backup/sweep
ls | /usr/lib/backup/copies > $tmp
for d in `sed 's:.* \\.*/.*:1:' $tmp | sort -u`
do
    if [ ! -d $d ]
    then
        echo mkdir $d 2>&1; mkdir $d
    fi
done
sed 's:~/usr/lib/backup/bcp -r : ' $tmp | sh -e
cut -f2 $tmp | /usr/lib/backup/fileupd | /usr/lib/backup/dbupdate

```

We first process the files with *sweep* assigning backup copy names, removing out of date copies, and either throwing away or complaining about bad files. The backup copy names have a simple volume/file structure, for example *v645/824*. The volumes (*v645*) are restricted to 10,000KB if possible. Next the files ready to be copied are extracted by *copies* whose output is holding-area-name, backup-copy-name pairs. After creating any missing volumes, the files are copied by *bcp*. This program differs from *cp(1)* in that it sets the *userid* and *groupid* to that of the original file and only allows destination filenames that look like backup copy filenames and have no links. (This is all in security's name; *munge* is executed by *root*.) After the files are set in their new home, *fileupd* extracts data so that *dbupdate* can add the version, backup copy pair to the database. An analogous program *wormupd* extracts data from files on a WORM.

The last stage consists of copying copies in the trusted area (called */backup/v*) onto archival media. Small systems, or systems who use magnetic tape, may postpone this step by using a trusted area spread over multiple file systems as described below. Our center backs up too much data to do this; instead, completed volumes are copied automatically onto optical disk. Each backup system will have different needs. The backup software supports these different methods by customising processing via the *PROCPERM* environment variable.

For example, we execute *munge* by

```
PROCFERH=/usr/lib/backup/toworm /usr/lib/backup/munge
```

Toworm simply goes through the backup copy volumes one at a time, writing them to optical disk and removing them (if no errors occurred).

This is not the right thing to do if you have magnetic tape rather than optical disk. We used magnetic tape for the first year or so of the backup system. We set aside a few file systems to store the backup copies and put symbolic links to the volumes in */backup/v*. Thus, */backup/v/v1456* might be a symbolic link to */disk2/v1456*. When we ran out of space, we simply dumped the (oldest) whole file system to tape via *dd(1)* and created more symbolic links. In this scenario, recovering old files means copying the relevant tape back onto a spare file system and constructing the appropriate symbolic links.

Periodically, the backup database should be copied onto backup media and compressed via the script */usr/lib/backup/bkdb*. As for *munge*, you can support your favourite media by writing your own script or if you have WORMs, you can use the supplied script */usr/lib/backup/wormdb* which simply copies the database onto optical disk. Small systems probably need to do this once every 1-2 months; large systems (like ours) every 2 weeks or so.

5. Experience

File backup is a game of high emotion. A successful recover of a lost file is often a moment of rare joy and relief. An unsuccessful recover, particularly when due to a screwup in the backup system, all too often leads to human sacrifice. The main goal throughout all the design and implementation has been reliability. The changes in architecture and growing size of the system are mainly a consequence of understanding more completely how badly the underlying operating system and network(s) can misbehave.

Batch mode The current design operates in a batch mode; all the candidate filenames are processed and then all the appropriate files are transported to the backup machine. The original design was more interactive; a filename was proposed and if appropriate, transported immediately. In some environments, this may be the best way but not in our systems. The databases we have do not support multiple writers and the interactivity implies very low bandwidth across our networks, mainly because of scheduling delays.

Why the first implementation failed The first implementation of the backup system offered almost the same functionality as the current version and yet was only 600 lines of C code. (The current source is 2400 lines, with another 1100 for WORM support.) It depended heavily on the network file system. Clients interrogated the database and then simply copied files into the trusted area. Unfortunately, it proved to be too unreliable. File system connections to other machines dropped frequently, administrative problems arose mapping userids between systems, and upon rare occasions, data was corrupted both by the remote file system code and the network. This is why the current system checksums all file transfers.

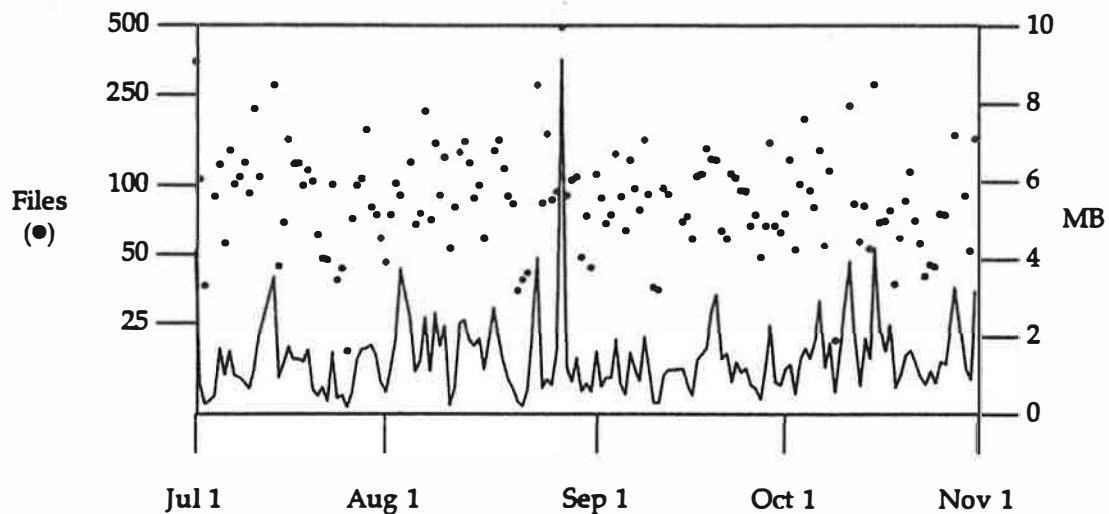
The current system uses the network file system for user database queries. However, all administrative communication to the backup system is now done by special servers which checksum all transferred data. Shortlived problems such as failed connections or other networking ills are simply ignored; the files are picked up the next night. This is a straight tradeoff between administrative overhead and the possibility to miss some shortlived files. Of course, if it really matters (say if the machine is an important one) simply run the *sel* ! act script again by hand.

The other hard lesson learned from the first implementation was to update the database only when you are really sure you have the backup copy. Too often, the files would get lost after they were put in the database but before they were archived. In the current system, the worst thing that can happen is to lose the backup copies that are not yet on

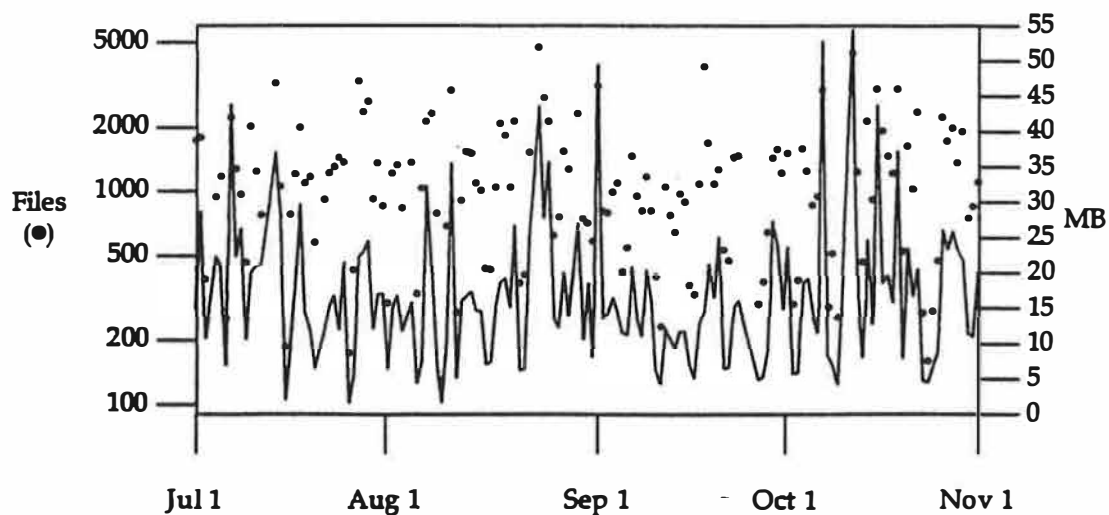
optical disk (probably because of some hardware failure). The database can be recovered in about 2-3 hours but the backup copies are gone (they will probably be picked up again the next night).

Data throughput The average number of files and bytes backed up has been surprisingly constant over the last eighteen months; roughly 30 files (315KB) per user per system per day. In general, though, you should plan for peak rates of three times this. Most often, high data rates can be anticipated and moderated by running the various scripts during the day. In any event, it is merely inconvenient to run out of space. Missed files will in general get picked up the next night.

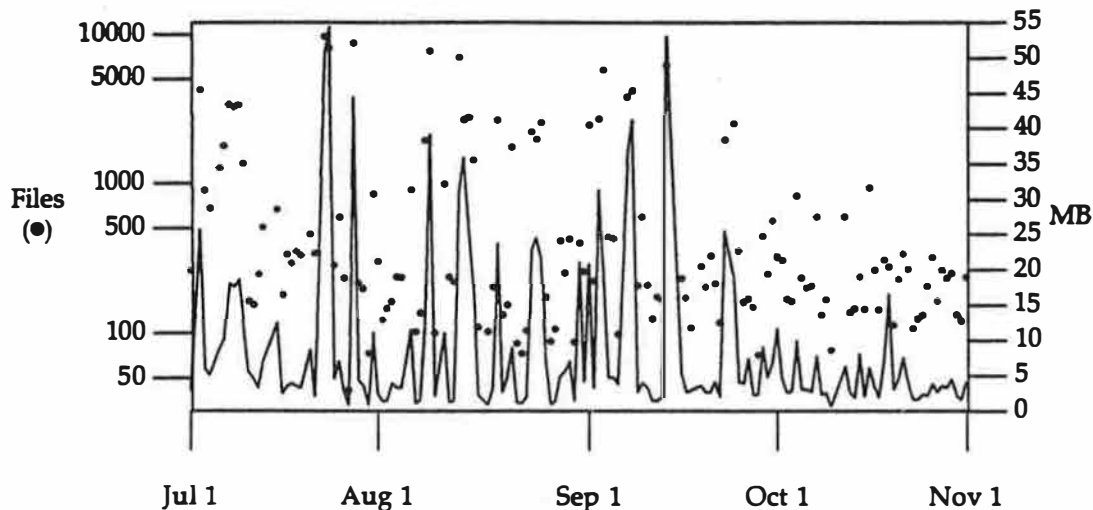
The throughput data for three systems are shown below. We show the number of files (●) and bytes backed up each day over a five month period. The system arend (a VAX 11/750) is a typical 3 user system:



The system coma (a VAX 8550) supports about 30 users. Its numbers are larger than arend because the system is new and subject to several reorganisations:



The final example is an atypical system `bowell`, which is our central source machine. It has no permanent users but is often very active. The average is just a little higher than normal but the variance is much higher.



Optical disks Our optical disks (SONY WDA-3000 CLV) have proved to be easy to use and reliable in operation. They make the backup system workable; they are easy for users to load and unload (about the same as a video cassette recorder), they have a huge capacity of 3.2G bytes per disk (so files are more likely to be immediately available), and are quite compact (one rack-wide shelf holds about 3 years of backup).

The file structure on the WORM is simple; it is *not* a normal disk file system. There are various commands that manipulate the file system including read, write, and initialise a WORM. It is a linked list of segments; each segment corresponds to a `worm mkfs` or `worm write` command. A segment has a header, a vector of pseudo-inodes (name, stat structure), a string table and the associated data blocks. The main cleverness is the (necessary) pre-allocating of the forward pointer to the next segment at the time you write a segment header.

Choice of database V9 Unix offers support for two types of large database: Ken Thompson's `dbm(1)` hash-table database and Peter Weinberger's `cbt(1)` compressed B-tree database. Both Thompson and Weinberger helped during the development of the backup software by fixing bugs and speeding up the routines. The original version of the software used `dbm`, the current version uses `cbt`.

There are three criteria used to compare the databases: size of the database, speed to probe for a given key, and speed to present every key in the database. Note that the `dbm` database is sparse; thus both size and disk usage is given. For a small database of 25989 `<key,value>` pairs we have

Criteria	dbm.4k	dbm.1k	cbt
size	.dir=109 .pag=3547136	.dir=506 .pag=4145152	.F=277337 .T=486400
disk	2009KB	2144KB	746KB
span	397.1u+44.4s=441.5	118.1u+16.6s=134.7	12.8u+86.8s=99.6
probe	63.8u+17.7s = 81.5	22.8u+12s=34.8	27.3u+10.1s=37.4

The `dbm.4k` figures are for the standard `dbm` software; `dbm.1k` has a smaller page size and some software speedups by Thompson. Times are given in user+system time. The probe times are for searching 2481 keys. To see how these results scaled up, I also compared

dbm.1k and cbt in a larger database of 242217 *<key,value>* pairs (9.3 times larger).

Criteria	dbm.1k	cbt
size	.dir=8049 .pag=65936384	.F=2733624 .T=3916800
disk	21690KB	6495KB
span	1186.5u+189.9s=1376.4	526.8u+581.7s=1108.5
probe	22.7u+14.5s=37.2	31.0u+16.6s=47.6

The probe times are for 2481 probes (same as above).

The comparison is fairly even but *cbt* has a significant edge in disk usage. This is important as the database will get large (several hundred megabytes). It is of course no surprise; the data being stored are absolute pathnames, perfectly suited to the leading substring compression that *cbt* uses. One criterion not mentioned above is experience with large databases. Thompson has done much chess endgame research with large (100–200MB) *dbm* databases. On the other hand, *cbt* has apparently never been used for large databases before. In fact, the backup database uncovered a bug limiting databases to 32MB. It would not be surprising if other (hidden) size restrictions show up.

In any case, after all the experimentation, it turns out that *dbm* can't be used anyway because of the restriction that all keys that hash together must fit on a single (internal) page. While a page overflow is unlikely, the workarounds are difficult.

Security Backup copies use the file system for access control. The access permissions for a backup copy are those of the original file with the write and execute bits turned off. Thus, the backup systems record permanently any mistakes such as creating a sensitive file with general read permission and fixing it a day later.

The only glaring security botch is that filenames, and in general pathnames, are always visible to all users even though they may not have been visible originally (say by an unreadable directory). This is not a problem in our environment; it may be in more paranoid systems.

Frequency of recovery We have noticed a great increase in the use of the backup system as file recovers get easier. At one site, file recovers increased from about one per week to about 5–6 per day. Some of this is real backup usage that wasn't done before because it wasn't convenient. The rest seems to be use of the backup system as an archival system; that is, deliberately removing files and recovering them when needed.

6. Future Work

The backup system works well for recovering a few files at a time. It is not so good at recovering large parts of a file system. After discussions with several people, notably John Linderman and Norman Wilson, it is clear that some relatively minor changes will allow the backup system to resurrect efficiently not just directories, but complete file trees, much like the Unix tools *dump(1)* and *restor(1)*. Another (small) set of changes would allow full and incremental dumps to be used as input to the backup system. This would be more efficient and easier to port to other systems which can generate *dump(1)* files.

We are integrating a 50 platter jukebox for our optical disks. The most obvious effect is to free users from mounting old disks. The other attractive feature is reduce administration overhead to reading occasional mail messages.

We are investigating moving to a more secure version of our Unix[2] so as to curb the ability of the superuser to delve into everybody's past files.

7. Conclusion

The backup system works, and works well. It is resilient against various failures such as network breakdowns, running out of space, and operator inattention. The backup files are entirely self-contained; the database can be (and has been twice) rebuilt from the backup files. It requires very little manual intervention, typically a little disk shuffling once every 2-4 weeks. With a jukebox, there is *no* manual intervention.

It is popular. Two other sites are running the current system; one of them, run by Ted Kowalski, even suffered through the earlier versions of the software. Three other sites are investigating using the system. The system is also portable to other Unix environments. Our system backs up a Sun file server directly.

Acknowledgements

Ken Thompson and Peter Weinberger were very supportive in fixing bugs and improving the speeds of their databases. Ted Kowalski has suffered longest in using the backup system; his users were the first real active customers. Paul Glick is the administrator for the other backup system.

References

- [1] Weinberger, P.J., *The Version 8 Network File System*, 1984 Summer (Salt Lake City) Usenix Conference Proceedings, p86.
- [2] Hume, A., *Grep Wars*, 1988 Spring (London) EUUG Conference Proceedings, p237.
- [3] McIlroy, M.D. & Reeds, J.A., *Multilevel Security with Fewer Fetters*, 1988 Spring (London) EUUG Conference Proceedings, p117.

SPIFF -- A Program for Making Controlled Approximate Comparisons of Files

Daniel Nachbar

Software Engineering Research Group
Bell Communications Research
Morristown, New Jersey

ABSTRACT

The well known program `diff` [1] is inappropriate for some common tasks such as comparing the output of floating point calculations where roundoff errors lead `diff` astray and comparing program source code where some differences in the text (such as white space and comments) have no effect on the operation of the compiled code. A new program, named `spiff`, addresses these and other similar cases by lexical parsing of the input files and then applying a differencing algorithm to the token sequences. `Spiff` ignores differences between floating point numbers that are below a user settable tolerance. Other features include user settable commenting and literal string conventions and a choice of differencing algorithm. There is also an interactive mode wherein the input texts are displayed with differences highlighted. The user can change numeric tolerances "on the fly" and `spiff` will adjust the highlighting accordingly.

Some Troubles With Diff

Over the past several years, it has been fairly easy to tell when a new type of computer arrived at a nearby computer center. The best clue was the discordant chorus of groaning, sighing, gnashing of teeth, pounding of foreheads on desks, and other sounds of distress. Tracing these noises to their source, one would find some poor soul in the process of installing a numerical analysis package on the new machine.

One might expect that "moving up" to a new machine would be a cause for celebration. After all, new machines are typically bigger, faster, and better than old machines. However, the floating point arithmetic on any new machine is frequently slightly different from any old machine. As a consequence, software package test routines produce output that is slightly different, but still correct, on the new machines. Serious troubles appear when the person installing the software package attempts to compare the test output files from two different machines by using a difference finding program such as `diff`. Programs such as `diff` do a character by character comparison. `Diff` finds a great many differences, most of which are due to roundoff errors in the least significant digits of floating point numbers. Others are the result of differences in the way in which the two test runs had printed a number (3.4e-1 vs. 0.34). In one case, the test suite for the S statistical analysis package[2], over 1700 floating point numbers are produced (per machine). In the eyes of `diff`, roughly 1200 of these numbers are different. However, none of the "differences" are important ones. Nonetheless, software installers wind up inspecting the output by eye.

A similar problem arises when one attempts to look for differences between two versions of the same C program. `Diff` reports many differences that are not of interest. In particular, white space (except inside quotation marks) and anything inside a comment have no effect on the operation of the compiled program and are usually not of interest. `Diff` does have a mode of operation where white space within a line (spaces and tabs) can be ignored. However, differences in the placement of newlines cannot be ignored. This is particularly annoying since C programming styles differ on whether to place a newline character before or after the '{' characters that start blocks.

The Problem in General Terms

As already mentioned, programs such as `diff` do a character-by-character comparison of the input files. However, when it comes to interpreting the contents of a file (either by a human or by a program) it is almost never the case that characters are treated individually. Rather, characters make up tokens such as words and numbers, or act as separators between these tokens. When comparing files, one is usually looking for differences between these tokens, not the characters that make them up or the characters that separate them.

What is needed is a program that first parses the input files into tokens, and then applies a differencing algorithm to the token sequences. In addition to finding differences in terms of tokens, it is possible to interpret the tokens and compare different types of tokens in different ways. Numbers, for example, can differ by a lot or a little.¹ It is possible to use a tolerance when comparing two number tokens and report only those differences that exceed the tolerance.

Design Issues

A serious design issue for such a program is how complex to make the parse. The deeper one goes in the parsing the larger the unit of text that can be manipulated. For instance, if one is looking for differences in C code, a complete parse tree can be produced and the differencing algorithm could examine insertion and deletion of entire branches of the tree. However, deep parsing requires much more complex parsing and slower differencing algorithms.

Another design issue is deciding how to interpret the tokens. Closer interpretation may lead to greater flexibility in comparing tokens, but also results in a more cumbersome and error-prone implementation.

In the program described here, we attempt to keep both the depth of the parse and the semantics of the tokens to a minimum. The parse is a simple lexical parse with the input files broken up into one dimensional sequences of numbers, literal strings and white space. Literal strings and white space are not interpreted. Numbers are treated as representing points on the real number line.

Default Operation

`Spiff`² works very much like `diff`. It reads two files, looks for differences, and prints a listing of the differences in the form of an edit script.³ As already suggested, `spiff` parses the files into literal strings and real numbers. The definition of these tokens can be altered somewhat by the user (more on this later). For now, suffice it to say that literals are strings like "cow", "sit", "into", etc. Real numbers look like "1.3", "1.6e-4" and so on. All of the common formats for real numbers are recognized. The only requirements for a string to be treated as a real number is the presence of a period and at least one digit. By default, a string of digits without a decimal point (such as "1988") is not considered to be a real number, but rather a literal string.⁴ Each non-alphanumeric character (such as `#$%^&*()`) is parsed into a separate literal token.

Once `spiff` determines the two sequences of tokens, it compares members of the first sequence with members of the second sequence. If two tokens are of different types, `spiff` deems them to be different, regardless of their content. If both tokens are literal tokens, `spiff` will deem them to be different if any of their characters differ. When comparing two real numbers, `spiff` will deem them to be different only if the difference in their values exceeds a user settable tolerance.

¹ Current differencing programs do not have such a notion because the difference between two characters is a binary function. Two characters are the same or they are not.

² We picked the name as a way to pay a small tribute to that famous intergalactic adventurer Spaceman Spiff[3]. `Spiff` is also a contraction of "spiffy diff".

³ An edit script is a sequence of insertions and deletions that will transform the first file into the second.

⁴ Integer numbers are often used as indices, labels, and so on. Under these circumstances, it is more appropriate to treat them as literals. Our choice of default was driven by a design goal of having `spiff` be very conservative when choosing to ignore differences.

Altering Spiff's Operation

To make *spiff* more generally useful, the user can control:

- how text strings are parsed into tokens
- how tokens of the same type are compared
- the choice of differencing algorithm used
- and the granularity of edit considered by the differencing algorithm.

These features are described next.

Altering the Parse

The operation of the parser can be altered in several ways. The user can specify that delimited sections of text are to be ignored completely. This is useful for selectively ignoring the contents of comments in programs. Similarly, the user can specify that delimited sections of text (including white space) be treated as a single literal token. So, literal strings in program text can be treated appropriately. Multiple sets of delimiters may be specified at once (to handle cases such as the Modula-2 programming language where there are two ways to specify quoted strings). At present, the delimiters must be fixed string (possibly restricted to the beginning of the line) or end of line. As a consequence of the mechanism for specifying literal strings, multicharacter operators (such as the `+=` operator in C) can be parsed into a single token.

As yet, no provision is made for allowing delimiter specification in terms of regular expressions. This omission was made for the sake of simplifying the parser. Nothing prevents the addition of regular expressions in the future. However, the simple mechanism already in place handles the literal string and commenting conventions for most well known programming languages.⁵

In addition to controlling literal string and comments, the user may also specify whether to treat white space characters as any other non-alphanumeric character (in other words, parse each white space character into its own literal token), whether to parse sign markers as part of the number that they precede or as separate tokens, whether to treat numbers without printed decimal markers (e.g. "1988") as real numbers rather than as literal strings, and whether to parse real numbers into literal tokens.

Altering the Comparison of Individual Tokens

As mentioned earlier, the user can set a tolerance below which differences between real numbers are ignored. *Spiff* allows two kinds of tolerances: absolute and relative. Specifying an absolute tolerance will cause *spiff* to ignore differences that are less than the specified value. For instance, specifying an absolute tolerance of 0.01 will cause only those differences greater than or equal to 0.01 to be reported. Specifying a relative tolerance will cause *spiff* to ignore differences that are smaller than some fraction of the number of larger magnitude. Specifically, the value of the tolerance is interpreted as a fraction of the larger (in absolute terms) of the two floating point numbers being compared. For example, specifying a relative tolerance of 0.1 will cause the two floating point numbers 1.0 and 0.91 to be deemed within tolerance. The numbers 1.0 and 0.9 will be outside the tolerance. Absolute and relative tolerances can be OR'ed together. In fact, the most effective way to ignore differences that are due to roundoff errors in floating point calculations is to use both a relative tolerance (to handle limits in precision) as well as an absolute tolerance (to handle cases when one number is zero and the other number is almost zero).⁶ In addition, the user can specify an infinite tolerance. This is useful for checking the format of output while ignoring the actual numbers produced.

⁵ See the manual page in the appendix for examples of handling C, Bourne Shell, Fortran, Lisp, Pascal, and Modula-2. The only cases that are known not to work are comments in BASIC and Hollerith strings in Fortran.

⁶ All numbers differ from zero by 100% of their magnitude. Thus, to handle numbers that are near zero, one would have to specify a relative tolerance of 100% which would be unreasonably large when both numbers are non-zero.

Altering the Differencing Algorithm

By default, *spiff* produces a minimal edit sequence (using the Miller/Myers differencing algorithm[4]) that will convert the first file into the second. However, a minimal edit sequences is not always desirable. For example, for the following two tables of numbers:

0.1	0.2	0.3	0.2	0.3	0.4
0.4	0.5	0.6	0.5	0.6	0.7

a minimal edit sequence to convert the table on the left into the table on the right be to would delete the first number (0.1) and insert 0.7 at the end.⁷ Such a result, while logically correct, does not provide a good picture of the differences between the two files. In general, for text with a very definite structure (such as tables), we may not want to consider insertions and deletions at all, but only one-to-one changes.⁸ So, rather than look for a minimal edit script, we merely want to compare each token in the first file with the corresponding token in the second file.

The user can choose which differencing algorithm to use (the default Miller/Myers or the alternative one-to-one comparison) based upon what is known about the input files. In general, files produced mechanically (such the output from test suites) have a very regular structure and the one-to-one comparison works surprisingly well. For files created by humans, the Miller/Myers algorithm is more appropriate. There is nothing in *spiff*'s internal design that limits the number of differencing algorithms that it can run. Other differencing algorithms, in particular the one used in *diff*, will probably be added later.

Altering the Granularity of the Edit Sequence

By default, *spiff* produces an edit sequence in terms of insertions and deletions of individual tokens. At times it may be more useful to treat the contents of the files as tokens when looking for differences but express the edit script in terms of entire lines of the files rather than individual tokens.⁹ *Spiff* provides a facility for restricting the edits to entire lines.

Treating Parts of the Files Differently

For complex input files, it is important that different parts of the file be treated in different ways. In other words, it may be impossible to find one set of parsing/differencing rules that work well for the entire file. *Spiff* can differentiate between parts of the input files on two bases: within a line and between lines. Within a line, a different tolerance can be applied to each real number. The tolerances are specified in terms of the ordinal position of the numbers on the line (i.e. one tolerance is applied to the first real number on each line, a different tolerance is applied to the second number on each line, a third tolerance is applied to the third, and so on). If more numbers appear on a line than there are tolerances specified, the last tolerance is applied to all subsequent numbers on the line (i.e., if the user specifies three tolerances, the third is applied to the third, fourth fifth, . . . number on each line). This feature is useful for applying different tolerances to the different columns of a table of numbers.

Between lines, the user can place "embedded commands" in the input files. These commands are instructions to parser that can change what tolerances are attached to real numbers and the commenting and literal string conventions used by the parser. Embedded commands are flagged to the parser by starting the line with a user-specified escape string. By combining within line and between line differentiation, it is possible for the user to specify a different tolerance for every single real number in the input files.

⁷ The problem of having the elements of tables become misaligned when the differencing algorithm is trying to find a minimal number of edits can be reduced somewhat by retaining newlines and not using tolerances. Unfortunately, it does not go away.

⁸ A "change" can be expressed as one deletion and one insertion at the same point in the text.

⁹ For instance, if one wants to have *spiff* produce output that can be fed into the *ed* editor.

Visual Mode

So far, **spiff**'s operation as an intelligent filter has been described. **Spiff** also has an interactive mode. When operating in interactive mode, **spiff** places corresponding sections of the input files side by side on user's screen.¹⁰ Tokens are compared using a one-to-one ordinal comparison, and any tokens that are found to be different are highlighted in reverse video. The user can interactively change the tolerances and **spiff** will alter the display to reflect which real numbers exceed the new tolerances. Other commands allow the user to page through the file and exit.

Performance

Two components of **spiff**, the parser and the differencing algorithm, account for most of the execution time. Miller and Myers compare their algorithm to the one used in the **diff** program. To restate their results, the Miller/Myers algorithm is faster for files that have relatively few differences but much slower (quadratic time) for files with a great many differences.

For cases where the files do not differ greatly, parsing the input files takes most of the time (around 80% of the total).¹¹ The performance of the parser is roughly similar to programs that do a similar level of parsing (i.e. programs that must examine each character in the file). For files where roughly half of the tokens are real numbers, **spiff** takes about twice as long to parse the input files as an **awk** program that counts the number of words in a file:¹²

```
awk '{total += NF}' firstfile secondfile
```

The time that it takes **spiff** to parse a file is substantially increased if scanning is done for comments and delimited literal strings. The precise effect depends upon the length of the delimiters, whether they are restricted to appear at beginning of line, and the frequency with which literals and comments appear in the input files. As an example, adding the 12 literal conventions¹³ and 1 commenting convention required for C code roughly doubles the time required to parse input files.¹⁴

A more complete approach to evaluating **spiff**'s performance must measure the total time that it takes for the user to complete a differencing task. For example, consider one of the test suites for the S statistical analysis package mentioned at the beginning of this paper. The output file for each machine is 427 lines long and contains 1090 floating point numbers. It takes **diff** approximately 2 seconds on one of our "6 MIPS"¹⁵ computers to compare the two files and produce an edit script that is 548 lines long containing 1003 "differences" in the floating point numbers. It takes the average tester 5 minutes to print out the edit script and roughly 2 hours to examine the output by hand to determine that the machines are, in fact, both giving nearly identical answers. The total time needed is 2 hours 5 minutes and 2 seconds.

In contrast, it takes **spiff** approximately 6 seconds on one of our "6 MIPS" computers to produce an output file that is 4 lines long.¹⁶ It takes the average tester 30 seconds to examine **spiff**'s output. The total for **spiff** is 36 seconds. Therefore for this case, **spiff** will get the job done roughly 208.88 times faster than **diff**.

¹⁰ Although the current implementation of **spiff** runs in many environments, interactive mode works only under the MGR window manager.[5] Other graphics interfaces will probably be added over time.

¹¹ No effort has yet been made to make the parser run more quickly. A faster parser could no doubt be written by generating a special state machine.

¹² For **awk**, a word is any string separated by white space.

¹³ One literal convention is for C literal strings. The rest enumerate multicharacter operators.

¹⁴ So in total, it takes **spiff** about 4 times longer to parse a C program than it takes **awk** to count the number of words in the same file.

¹⁵ We will not comment on the usefulness of "MIPS" as a measure of computing speed. The numbers provided are only intended to give the reader some vague idea of how fast these programs run.

¹⁶ The output would be zero length except that the output of the time command is built into the S tests. The timing information could easily be ignored using **spiff**'s embedded commands. But, as we shall see, it hardly seems worth the trouble.

In general, it is misleading to compare `spiff`'s speed with that of `diff`. While both programs are looking for differences between files, they operate on very different types of data (tokens vs. bytes). An analogous comparison could be made between the speed of an assembler and the speed of a C compiler. They are both language translators. One runs much faster than the other. None the less, most programmers use the slower program whenever possible.

Using Spiff For Making Regression Tests Of Software

We envision `spiff` to be the first of several tools for aiding in the now arduous task of making regression tests.¹⁷ Given `spiff`'s current capabilities, the regression test designer can take the output of an older version of software and through the use of literal string and commenting conventions, specify what parts of the output must remain identical and what sections can change completely. By specifying tolerances, the test designer can take into account how much of a difference in floating point calculations is acceptable.

The test designer is also free to edit the output from the older version of the software and add embedded commands that can instruct `spiff` to treat various parts of the output differently. The newly edited output can then serve as a template for the output of later versions of the software.

Obviously, editing output by hand is a very low level mechanism for adding specification information. It is our intention that `spiff` will become the last element in a pipeline of programs. Programs (as yet unwritten) located earlier in the pipeline can implement a higher level representation of the specification information. They read in the old and new input files, add the appropriate embedded commands, and then pass the results to `spiff` which will do the actual differencing.

Future Work

There are many features that could be added to `spiff` (if there are not too many already). Some of these include:

- Using separate differencing algorithms on separate sections of the file and/or limiting the scope of an edit sequence (fencing)
- Providing a more general mechanism for specifying comments and literals (perhaps allowing specification in terms of regular expressions). As yet, we have not encountered any important cases where regular expressions have been needed. Until such a case is encountered, we will leave regular expressions out in the name of simplicity.
- Allowing for a more general specification of what lines should look like. At present, the user can only specify tolerances for numbers as a function of their ordinal position on a line. The difficulty in expanding the specification abilities of `spiff` is knowing when to stop. In the extreme, we might add all of the functionality of a program such as `awk`.¹⁸ We hope to keep `spiff` as simple as possible. Our first efforts in this direction will try to implement higher level specification functions outside of `spiff`.

Acknowledgements

First and foremost, we thank Stu Feldman for his endless patience, constant encouragement and numerous good ideas. We also extend thanks to Doug McIlroy for bringing the Miller/Myers algorithm to our attention, Nat Howard for a key insight and for his editorial comments and Steve Uhler and Mike Bianchi for their editorial comments.

¹⁷ In software engineering parlance, a "regression test" is the process by which a tester checks to make sure that the new version of a piece of software still performs the same way as the older versions on overlapping tasks.

¹⁸ Imagine handling the case such as "apply this tolerance to all numbers that appear on a line starting with the word 'foo' but only if the number is between 1.9 and 3.6 and the word 'bar' does not appear on the line".

References

- [1] Hunt, J.W. and M.D. McIlroy. *An Algorithm For Differential File Comparisons*, Bell Labs Computer Science Technical Report, Number 41, 1975.
- [2] Becker, R.A. and J.M. Chambers (1984). *S - An Interactive Environment For Data Analysis And Graphics*. Belmont, CA: Wadsworth Inc.
- [3] Watterson, B. (1987). *Calvin and Hobbes*. New York: Andrews, McMeel & Parker.
- [4] Miller, W. and E.W. Myers. *A File Comparison Program, Software - Practice and Experience* 15, 11, 1025-1040, 1985.
- [5] Uhler, S.A. *MGR -- A Window Manager For UNIX*, Sun User's Group Meeting. September 1986.

SPIFF(1)

UNIX Programmer's Manual

NAME

spiff - make controlled approximate comparisons between files

SYNOPSIS

spiff [*-s script*] [*-f sfile*] [*-blevniqcdwm*] [*-a|-r value*] *-value file1 file2*

DESCRIPTION

Spiff compares the contents of *file1* and *file2* and prints a description of the important differences between the files. White space is ignored except to separate other objects. *Spiff* maintains tolerances below which differences between two floating point numbers are ignored. Differences in floating point notation (such as 3.4 3.40 and 3.4e01) are treated as unimportant.

Spiff's operation can be altered via command line options, a command script, and with commands that are embedded in the input files.

The following options affect *spiff's* overall operation.

- n* suppresses all output (but the return code may still be examined).
- q* suppresses warning messages.
- v* use a visually oriented display. Works only in mgr windows.

The following option controls the unit of change used by the differencing algorithms.

- l* consider only insertion/deletion of entire lines when using Miller/Myers. This option gives output that is somewhat similar in appearance to the output of *diff(1)*.

The following option controls the differencing algorithm.

- e* compare each object (token or line as specified by the presence or absence of the *-l* flag) in the files with the object in the same ordinal position in the other file. If the files have a different number of objects, a warning message is printed and the objects at the end of the longer file are ignored. By default, *spiff* uses a Miller/Myers algorithm to find a minimal edit sequence that will convert the contents of the first file into the second.

<decimal-value>

sets a limit on the total number of insertions and deletions that will be considered. If the files differ by more than the stated amount, the program will give up, print a warning message, and exit.

The following options control the command script. More than one of each may appear at a time. The commands accumulate.

- f sfile* a command script to be taken from file *sfile*
- s command-script*
causes the following argument to be taken as a command script.

The following options control how individual objects are compared.

- b* treat all objects (including floating point numbers) as literals.
- c* ignore differences between upper and lower case.

The following commands will control how the files are parsed.

- w* treat white space as objects. Each white space character will be treated as a separate object when the program is comparing the files.
- m* treat leading sign characters (+ and -), as separate even if they are followed by floating point numbers.

SPIFF(1)

UNIX Programmer's Manual

- d** treat integer decimal numbers (such as 1987) as real numbers (subject to tolerances) rather than as literal strings.

The following three flags are used to set the default tolerances. The floating-point-numbers may be given in the formats accepted by `atof(3)`.

- a floating-point-number**

specifies an absolute value for the tolerance in floating point numbers. The flag **-a1e-2** will cause all differences greater than or equal to 0.01 to be reported.

- r floating-point-number**

specifies a relative tolerance. The value given is interpreted as a fraction of the larger (in absolute terms) of the two floating point numbers being compared. Thus, the flag **-r0.1** will cause the two floating point numbers 1.0 and 0.91 to be deemed within tolerance. The numbers 1.0 and 0.9 will be outside the tolerance.

- i** causes differences between floating point numbers to be ignored.

If more than one **-a**, **-r**, or **-i** flag appear on the command line, the tolerances will be OR'd together (i.e. any difference that is within any of the tolerances will be ignored).

SCRIPT COMMANDS

A script consists of commands, one per line. Each command consists of a keyword possibly followed by arguments. Arguments are separated by one or more tabs or spaces. The commands are:

literal BEGIN-STRING [END-STRING [ESCAPE-STRING]]

Specifies the delimiters surrounding text that is to be treated as a single literal object. If only one argument is present, then only that string itself is treated as a literal. If only two arguments are present, they are taken as the starting and ending delimiters respectively. If three arguments are present, they are treated as the start delimiter, end delimiter, and a string that may be used to escape an instance of the end delimiter.

beginchar BEGINNING-OF-LINE-CHARACTER

Set the the beginning of line character for BEGIN-STRING's in comments. The default is '^'.

endchar END-OF-LINE-CHARACTER

Set the end of line character for END-STRING's in comments. The default is '\$'.

comment BEGIN-STRING [END-STRING [ESCAPE-STRING]]

Specifies the delimiters surrounding text that is to be ignored entirely (i.e. viewed as comments). The operation of the comment command is very similar to the literal command. In addition, if the END-STRING consists of only the end of line character, the end of line will delimit the end of the comment. Also, if the BEGIN-STRING starts with the beginning of line character, only lines that begin with the BEGIN-STRING will be ignored.

More than one comment specification and more than one literal string specification may be specified at a time.

resetcomments

Clears the list of comment specifications.

resetliterals

Clears the list of literal specifications.

SPIFF(1)

UNIX Programmer's Manual

tol [**a**VALUE|**r**VALUE|**i**|**d**...[; **a**VALUE|**r**VALUE|**i**|**d**...]...

set the tolerance for floating point comparisons. The arguments to the **tol** command are a set of tolerance specifications separated by semicolons. If more than one **a**, **r**, **d**, or **i** appears within a specification, then the tolerances are OR'd together (i.e. any difference that is within any tolerance will be ignored). The semantics of **a**, **r**, and **i** are identical to the **-a**, **-r**, and **-i** flags. The **d** means that the default tolerance (as specified by the invocation options) should be used. If more than one specification appears on the line, the first specification is applied to the first floating point number on each line, the second specification to the second floating point number on each line of the input files, and so on. If there are more floating point numbers on a given line of input than tolerance specifications, the last specification is used repeatedly for all remaining floating point numbers on that line.

command STRING

lines in the input file that start with STRING will be interpreted as command lines. If no "command" is given as part of a **-s** or **-f** then it will be impossible to embed commands in the input files.

used to place remarks (comments) into a commands script.

Tolerances specified in the command scripts have precedence over the tolerance specified on the invocation command line. The tolerance specified in *file1* has precedence over the tolerance specified in *file2*.

VISUAL MODE

If *spiff* is invoked with the **-v** option, it will enter an interactive mode rather than produce an edit sequence. Three windows will be put on the screen. Two windows will contain corresponding segments of the input files. Objects that appear in both segments will be examined for differences and if any difference is found, the objects will be highlighted in reverse video on the screen. Objects that appear in only one window will have a line drawn through them to indicate that they aren't being compared with anything in the other text window. The third window is a command window. The command window will accept a single tolerance specification (followed by a newline) in a form suitable to the **tol** command. The tolerance specified will then be used as the default tolerance and the display will be updated to highlight only those objects that exceed the new default tolerance. Typing **m** (followed by a newline) will display the next screenfull of text. Typing **q** (followed by a newline) will cause the program to exit.

EXAMPLES

spiff -l -a1e-5 -r0.001 foo bar

compare the contents of the files *foo* and *bar* and ignore all differences between floating point numbers that are less than 0.00001 or 0.1% of the number of larger magnitude and report the differences in terms of whole lines of the files rather than individual tokens.

spiff -w -l -b foo bar

will produce output very much like the output of **diff(1)**.

tol a.01 r.01

will cause all differences between floating point numbers that are less than 0.01 or 1% of the number of larger magnitude to be ignored.

tol a.01 r.01 ; i

will cause the tolerance in the previous example to be applied to the first floating point number on each line. All differences between the second and subsequent floating point numbers on each line will be ignored.

SPIFF(1)

UNIX Programmer's Manual

tol a.01 r.01 ; i ; a.0001

like the above except that only differences between the second floating point number on each line will be ignored. The differences between third and subsequent floating point numbers on each number will be ignored if they are less than 0.0001.

A useful script for examining C code is:

```
literal " " \
comment /* */
literal &&
literal |
literal <=
literal >=
literal !=
literal ==
literal -
literal ++
literal <<
literal >>
literal ->
```

A useful script for shell programs is:

```
literal ' ' \
comment # $
```

A useful script for Fortran programs is:

```
literal ' ' '
comment ^C $
```

A useful script for Modula 2 programs is:

```
literal ' '
literal " "
comment (* *)
literal :=
literal <>
literal <=
literal >=
```

A useful script for Lisp programs is:

```
literal " "
comment ; $
```

DIAGNOSTICS

*Spi*ff's exit status is 0 if no differences are found, 1 if differences are found, and 2 upon error.

BUGS

In C code, escaped newlines will appear as differences.

Comments are treated as token delimiters.

Comments in Basic don't work right. The line number is not ignored.

Continuation lines in Fortran comments don't work.

There is no way to represent strings specified using a Hollerith notation in Fortran.

SPIFF(1)

UNIX Programmer's Manual

In formatted English text, hyphenated words, movements in pictures, footnotes, etc. will be reported as differences.

STRING's in script commands can not include whitespace.

Visual mode does not handle tabs properly. Files containing tabs should be run through expand(1) before trying to display them with visual mode.

In visual mode, the text windows appear in a fixed size and font. Lines longer than the window size will not be handled properly.

Objects (literal strings) that contain newlines cause trouble in several places.

Visual mode should accept more than one tolerance specification.

When using visual mode or the exact match comparison algorithm, the program should do the parsing on the fly rather than truncating long files.

AUTHOR

Daniel Nachbar

SEE ALSO

atof(3) diff(1) expand(1) mgr(1L)

"Spiff -- A Program for Making Controlled Approximate Comparisons of Files", by Daniel Nachbar.

"A File Comparison Program" by Webb Miller and Eugene W. Myers in Software -- Practice and Experience, Volume 15(11), pp.1025-1040, (November 1985).

Rtools: Tools for Software Management

in a

Distributed Computing Environment

Helen E. Harrison, Stephen P. Schaefer, Terry S. Yoo

Microelectronics Center of North Carolina

Post Office Box 12889

Research Triangle Park, NC 27709

ABSTRACT

We have implemented a software management tool-set which addresses the problems of compilation and distribution of software across multiple architectures in a mixed-vendor distributed environment. This package, which we call *rtools*, is made up of a central database, a compilation tool, and a distribution tool. We describe the system in detail and look at possible implementations of *rtools* in other environments. The system is currently managing software which spans three operating system versions on two architectures and twenty-two hosts.

1. Introduction

Rtools, an automated software management system, relieves the tedious and sometimes difficult chores of recompiling and updating large quantities of software in a heterogeneous distributed environment. An environment that includes a mix of architectures, a large quantity of source code to maintain, and a large number of machines presents problems which are different from those in an environment with a smaller number of machines. There are many approaches to automating software maintenance. We have tried several of them and have had greater success with those approaches that did not require modification of the individual software subsystems. In implementing the *rtools* tool-set we endeavored to improve the overall administration of software and facilitate the handling of object and executable files, while maintaining the integrity of the existing system.

A First Approach Using Makefiles

A first cut at automating the recompile and redistribute process used *make(1)*.¹ Makefiles were modified to accommodate compilation for multiple architectures. They also contained targets to distribute executable files to all appropriate machines. A standard template for the enhanced makefile structure was used to aid programmers in managing the large amounts of additional information introduced by the problems of remote compilation and distribution.

To aid in remote installation, we originally wrote *rinstall*, a utility whose operation was

much like *install(1)*. *Rinstall* was a front end to *rdist*, the Berkeley UNIX[™] file distribution utility. In addition to the usual facilities of *install*, *rinstall* allowed one to specify a list of machines upon which to install a file.

Problems

The makefiles used in this approach were complex and difficult to maintain. We had to rewrite every makefile to include the ability to compile and install programs on machines which may have been neither operating-system nor binary compatible with the source machine. For instance, the old makefile for the well known public domain program *shar* was over a hundred lines and 2K characters long for a program with only two C source files and two manual pages. At that size, it supported only two operating systems. The effort required in the recent acquisition of Ultrix[™] as a supported system exposed many shortcomings in using makefiles as the control point for file creation and distribution.

Furthermore, *rinstall* proved insufficient for our needs. It worked in a linear manner, taking one file and installing it on several machines. *Rinstall* did not take full advantage of the power available in the Berkeley *rdist* utility. It did not check modification dates, nor did it transfer multiple files over the same connection.

What was adequate for an environment of four or five machines running a single implementation of UNIX, is not sufficient for our growing environment of over thirty hosts of several architectures and five versions of UNIX.

The Solution

Our objective was to unite existing tools into a usable package that used the strengths of those tools to the best advantage.

We recognized that *make* was never intended to be used to control the management of software at the level we required. Its strengths lie in its ability to efficiently and correctly create binary files. To alleviate the inadequacies of *make* for our task, we separated the problem into two distinct tasks: creating binaries and installing them.

Likewise *rdist* is generally adequate for putting several objects on several machines. However, *rdist* has some of the same maintenance requirements as *make*. In particular, it requires that a separate control file be created for every distribution directive. In order to circumvent the problems of maintaining large numbers of control files, we specify a central point of distribution control.

The result is *rtools*: a system built around *make* and *rdist*, which separates compilation and distribution. With *rtools* makefiles need only create executables; other means are used to distribute and install them. *Rtools* creates and installs binaries based on information in a central database, hiding the details of remote compilation and installation for other architectures from the user and from the makefiles.

UNIX is a registered trademark of AT&T.

ULTRIX is a registered trademark of Digital Equipment Corporation.

Design Overview of Rtools

Splitting the work along the functional lines of the file creation process and the file distribution process is a conceptual step above the previous method of handling both creation and distribution in the makefiles of each source code subsystem. The major elements associated with this system are: a database which maintains necessary software management information, *rcreate* which handles the creation/compilation of object/executable files, and *rput* which deals with the distribution of those same files (see figure 1).

In essence, *rcreate* and *rput* are simple programs that drive the individual system elements. Each calls tools that extract information from the database and pass that information to the intermediate tools which in turn process the database information into directives for standard UNIX facilities. From there, the creation or distribution process is completed using well known UNIX tools, drawing on the information provided by the *rtools* system.

The system employs existing features where possible, taking advantage of the pipes and UNIX 4.3BSD tools to avoid re-inventing capabilities which exist in utilities such as *make* or *rdist*. The tool-set uses standard input and standard output for communication and depends heavily on the Bourne Shell for internal cohesiveness.

2. Shadow Trees

For storage of the distributable files, we create "object trees" which mimic the organization of a source code directory, for example: `/usr/src` (see figure 2). There is one object tree for each different architecture or operating system. We called the directories where object files are stored, "Shadow Trees", because their structure is a projection of the source code tree.

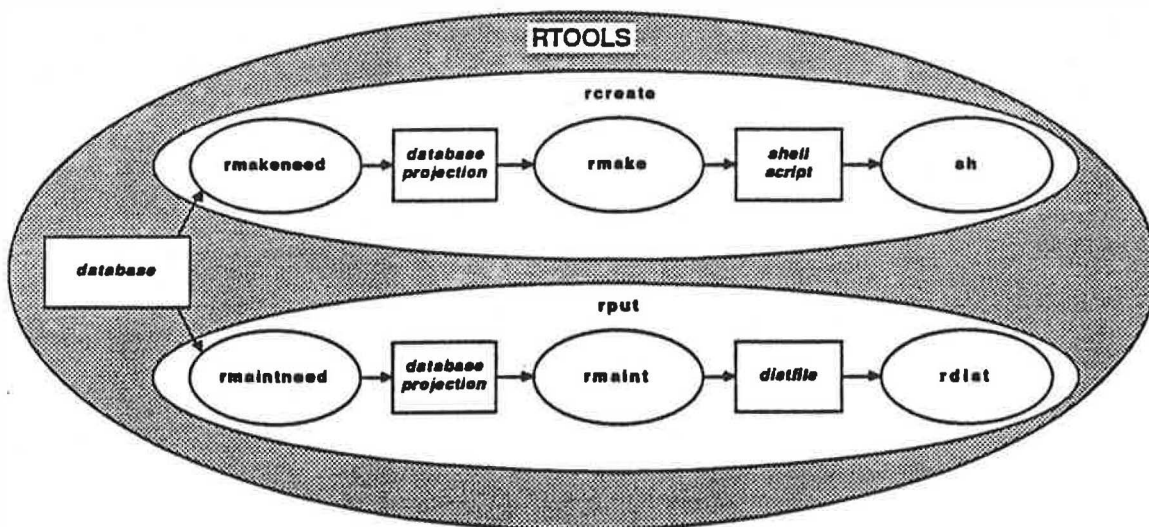


Figure 1 - A representation of the *Rtools* design

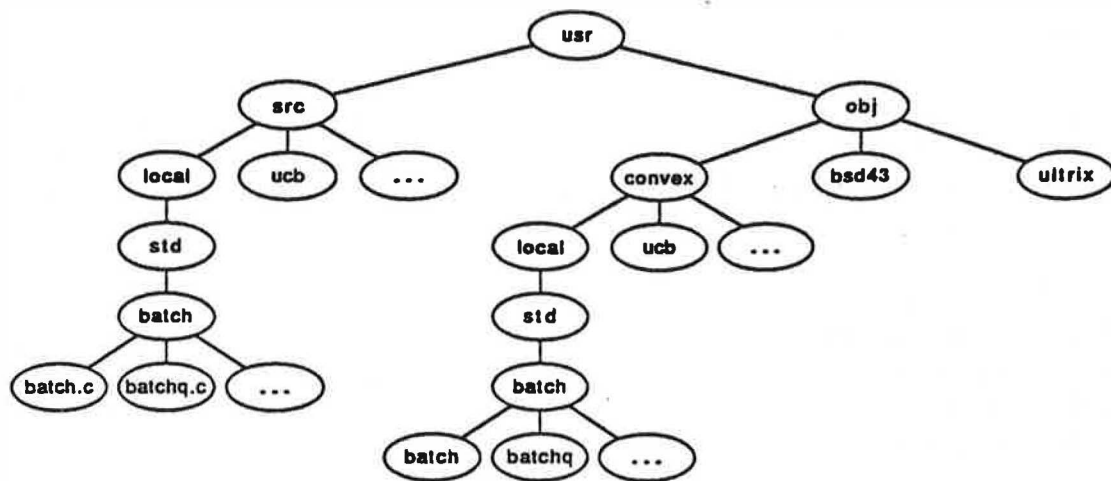


Figure 2 - Rtools Directory Structure

A holding directory contains all the object trees (or "shadow trees"); this holding directory is attached at the same level as the original source directory. The holding directory is `/usr/obj` in the example below. The correspondence of the trees determines exactly where the executable for a particular architecture may be found, given the location of its source code. For example, the source code for a program called *batch*, might reside in the file `/usr/src/local/std/batch/batch.c`. The resulting object file, compiled for a Convex C-1 computer would be the file: `/usr/obj/convex/local/std/batch/batch.o`.

3. Control Database

All the information for remote compilation and distribution formerly kept in makefiles is now centrally located in a single control database. A reason for using a single central database is that some information is common to both the compilation and the distribution process. Furthermore, by removing that information from the makefiles *rtools* is able to abstract the distributed nature of the environment away from the compilation tools.*

In hindsight, this is a simple concept. It is also easy to implement but will be difficult to maintain as the database grows. Until proper database support tools are devised, some aspects of maintaining the information will remain awkward.

*The Tektronics "Utek Build Environment" makes similar use of a central database for its source management, but with a different goal. Their system provides a software development environment, while *rtools* is designed for software maintenance.²

Records

There is one record for each distributed file (see table 1). Each record in the control database is divided into ten fields. The data in the fields control the selection and utilization of individual records. Each record can be selectively accessed through one of two data extraction tools. They mask away information unnecessary for some particular step in the distribution or installation process.

Fields

Each record contains the following fields:

- **Host List**
This field contains a *host list*, describing the systems for which the current record applies.
- **Installation Class**
Currently, the installation class is one of "bin", "lib" or "man". which correspond respectively to executables, library files, and man pages. Other classes or the capability for multiple classes may be appropriate as the database develops.
- **Target Architecture**
The architecture or operating system for which this file is to be prepared.
- **Shadow Directories**
Distributed objects/executables reside in a directory subtree appropriate to their target architecture. This field denotes the path for both the source directory and the object directory for the file described by the current record.
- **Make Target**
This field contains the target that directs *make* to create the file locally, in preparation for distribution.
- **Local File Name**
Usually identical to the *make* target
- **Remote Destination Name**
This field is the fully qualified (e.g. /usr/local/gnu/bin) path name of the destination

Field	Used In	Example
Host List	file creation & distribution	natasha boris
Target Architecture	file creation & distribution	convex
Installation Class	file creation & distribution	bin
Shadow Directory	file creation & distribution	local/std/batch
Make Target	file creation only	batch
Local File Name	file distribution only	batch
Remote Destination Name	file distribution only	/usr/local/std/bin/batch
Owner	file distribution only	bin
Group	file distribution only	bin
Mode	file distribution only	755

Table 1 - A database record

directory and filename on the remote host.

- Owner, Group, and Mode Fields

These three fields specify the eventual owner, group designation and file permission mode on the remote machines.

4. Rcreate: The Binary Creation Tool

Rcreate, the *rtools* compilation utility, must accomplish three tasks. First, it must acquire information about the current location of source files and the final destination of the executables. Through *rmakeneed*, a data access program, the database provides all the necessary information. Second, it must compile software for remote machines, accommodating different operating systems and different architectures. The final requirement is that it place all object and executable files within the appropriate shadow tree.

In *rtools* we made no additional enhancements to *make*. Instead we augment the make process with a superstructure of shell directives. *Rcreate* makes use of our local distributed computing utilities. *FREEDOMNET*,^{3,4} developed at the Research Triangle Institute, is a distributed computing system that transparently provides both remote execution and remote file access across heterogeneous architectures. We use *FREEDOMNET* not only for maintaining sources in one virtual place, but also for remote compilation on different architectures. Specifically for its second task, *rcreate* uses the *FREEDOMNET* utility *excr* which makes the process's notion of the root directory (*/*) relative to the remote machine. *Excr* preserves all of the current environment, including the current working directory, while all of the programs invoked, in particular compilers, execute on the foreign machine.

To accomplish the third task, *rcreate* employs *build*, a *make* utility developed at AT&T Bell Laboratories and reimplemented at the Research Triangle Institute.⁵ *Build* is a "view path searching version of *make*."⁶ It uses the environment variable 'VPATH' which specifies the directory trees to search when looking for makefiles and source files. It also maintains the VPATH variable through recursive invocations in subdirectories. It also has the ability to augment the VPATH variable dynamically as scripts within the makefile change directories and again invoke *make/build*. *Rcreate* uses this mechanism to maintain the separation between the source and object.

Rcreate itself is a pipeline of other tools: *rmakeneed*, the database data extraction utility; *rmake*, the compilation coordinator; and the Bourne shell to invoke *build*.

Rmakeneed

Rmakeneed is a database projection tool. It bridges the information in the database with the a script generator and serves as the source for the compilation pipeline. Its function is to extract the information required to create files for machines throughout the distributed environment. The output has the form of directives for the *rmake* utility.

Rmake

It is in the *rmake* step that the distributed nature of the environment is introduced into the make process. *Rmake* takes the database projection from *rmakeneed* and produces a shell script with the proper environment variables, etc. to *excr* to a machine of the appropriate architecture and run *build* for each of the specified *make* targets. It should be stressed that the "make-ing" utility is not affected by this step.

Build

Build is called when the shell script produced by *rmake* is run. The previous steps of creating an ad hoc compilation environment allows *build* to proceed in a normal fashion, oblivious to the nature of the distributed environment in which it is working.

5. Rput: The Distribution Tool

Rput is the *rtools* file distribution utility. It consists of a pipeline, beginning with *rmaintneed*, which extracts only the information necessary and sufficient for a particular distribution from the *rtools* database. Those results are piped to *rmaint*, which reformats the information into instructions for the *rdist* utility. *Rdist* instructions are a specification of what files on the current host should be put at what file names on which hosts. These instructions are commonly held in a file called a "distfile." In this instance, however, the instructions are piped directly to *rdist*. *Rtools* needs only a well defined subset of the the extensive capabilities of *rdist*; by generating distfiles ad hoc, we avoid maintenance of many hundreds of distfiles containing information mostly redundant with respect to each other or with respect to the contents of the control database.

Rmaintneed

Rmaintneed is a database access tool whose output includes directives for *rmaint*. Like *rmakeneed*, *rmaintneed* serves as the head of its process pipeline. However, the information extracted by *rmaintneed* is specific to the distribution process.

Rmaint

The labor saving tool of the distribution process is *rmaint*. *Rmaint* automatically generates distfiles for the *rdist* program, hiding the details of *rdist* syntax from *rmaintneed*. It takes distribution specifications on its stdin and puts the distfile on its stdout.

Rdist

The actual distribution of files is performed by *rdist*. This powerful tool relieves much of the burden of verifying modification dates, connecting to remote machines, and handling the security issues surrounding distributed environments.

6. Rtools in Other Environments

We consider the design of the system to be more useful than any particular program comprising it. The shell scripts are almost trivial, but they obviously rely heavily on some powerful tools that were already implemented. A similar system could be implemented for a different environment using other commonly available tools. Here we discuss some of the details of implementing elements of the *rtools* package in other environments.

Distributed File Systems

Perhaps the most striking feature used by the *rtools* package is the FREEDOMNET *excr* utility. We surmise that *Excr* could be emulated in other distributed file systems by mounting directories of the source machine onto a target machine, and then invoking a compiler on the target machine, perhaps with *rsh* (Berkeley remote shell).

In the absence of a remote file system, some elements of the *rtools* design could be implemented using other UNIX concepts (e.g. remote daemons for compilation and *tar* or *cpio* for file transfer). One might also examine the traditional alternative of using cross compilers.

Using Make/Build

Build is more subtly important: it allows a minimum of intervention into a working makefile to adapt it to a variety of situations. The *make* delivered with 4.3BSD also recognizes the *VPATH* variable but without the ability to change when invoked in another directory. Such recursive calls of *make* happen surprisingly often — it appears in over fifty of the software subsystems delivered with source to 4.3BSD. *Nmake*, a fourth generation make utility in the AT&T Toolchest,⁷ may also cope with this situation.

Using Rdist

Rput depends heavily on *rdist*. In its absence, one might implement a work-alike using *rsh* (the 4.3BSD remote shell command) and *tar*, but there are important considerations that *rdist* addresses properly, such as having the loop over machines outside the loop over files, and the network security issues involved in a utility that will install sensitive programs (and what program isn't?).

We should mention here a local change to *rdist*: when the intended user on the remote host has not given permission for the user at the local host to gain access automatically with an appropriate *.rhosts* file, our version of *rdist* will prompt for the appropriate password. The behavior of the Berkeley version of *rdist* was to simply deny permission.

We also added options to allow more flexibility in maintaining remote files, namely set owner, group, and mode, do comparisons, and install even if the file has not been modified.

Information Management

We chose a database format that required the least possible work in creating database tools. After more experience with the system, we may decide that more formal and better controlled database tools would be worth the investment, at which point we could turn to a traditional DBMS to manage the data and to produce the input expected by *rmake* and *rmaint*.

7. Conclusions

Rtools is in production use and is currently managing software in an environment that includes 4.3BSD, Ultrix, and Convex 4.1 UNIX implementations. We plan to add *rtools* support for Sun UNIX 3.5 in the immediate future, and possibly Apple UNIX later.

We have a powerful tool with simple syntax after a modest programming effort. It succeeds even with complicated software systems with many machine dependencies and large numbers of components, such as GNU *emacs*. It has automated an error-prone and tedious task, allowing us to support a consistent environment in a more timely and reliable manner. It frees support staff to concentrate their attention elsewhere.

Background

The Microelectronics Center of North Carolina is a non-profit corporation involved in cooperative research and education that unites the resources of five participating universities, the Research Triangle Institute and the MCNC central research laboratory.

MCNC computer facilities at the central laboratory include 2 VAX 8650's, one VAX 11/750 and an assortment of Microvax II's running 4.3BSD UNIX 2 Convex C-1's (4.2BSD variant), several Vaxstation 2000 and Microvax GPX's running Ultrix 2.2, and a Sun-3/260 (Sun UNIX 4.2 v3.5) all networked together via ethernet. In the coming years, we expect a great increase in the number of workstations.

References

1. Helen E. Harrison, "Maintaining a Consistent Software Environment," *Proceedings of the Large Installation System Administrators Workshop*, (April, 1987).
2. Alan McIvor, "UTek Build Environment," *Proceedings of the Summer USENIX Conference*, pp. 437-443 (1987).
3. Bob Warren, Tom Truscott, Kent Moat, and Mike Mitchell, "Distributed Computing using RTI's FREEDOMNET in a Heterogeneous UNIX Environment," *Proceedings of the UNIFORM Conference*, pp. 115-126 (1987).
4. Tom Truscott, Bob Warren, and Kent Moat, "A State-Wide UNIX Distributed Computing System," *Proceedings of the Summer USENIX Conference*, pp. 499-513 (1986).
5. Kent Moat, "Design of build: A Path Searching make," Technical Memorandum, Research Triangle Institute., Research Triangle Park, NC 27709 (1986).
6. Verlyn Erickson and John Pellegrin, "Build - A Software Construction Tool," *AT&T Bell Laboratories Technical Journal*, (July-August, 1984.).
7. Glenn S. Fowler, "The Fourth Generation Make," *Proceedings of the Summer USENIX Conference*, pp. 159-174 (1985).

Tools and Policies for the Hierarchical Management of Source Code Development

Thomas Lord

Information Technology Center, Carnegie Mellon University
lord+@andrew.cmu.edu

ABSTRACT

This paper presents a brief description of the problems faced by the Information Technology Center as it attempts to distribute its software to a wide audience. An analysis of the difficulties concludes that (1) the development process suffers from poor communication between programming groups and (2) better programming tools are needed than just RCS and make.

To solve the first problem, we present an organizational model called hierarchical software management. The techniques described can greatly reduce the amount of last minute work that goes into producing a deliverable system.

To solve the second problem, we have designed replacements for make and RCS. The construction system has been implemented and is described in detail. By means of dynamic dependency tracking and automatic script generation, our make replacement relieves programmers of almost all the work normally associated with maintaining a *make* script. The RCS replacement has not been implemented yet, but is briefly presented in order to further explain relationship to hierarchical software management.

1. Introduction

This paper is about the authors' ongoing experience in bringing software management to a research organization of about 30 programmers at the Information Technology Center of Carnegie Mellon University. The ITC, which has for two years been providing an integrated UNIX[†] workstation computing environment to the CMU campus is becoming well known for its contribution of the Andrew Toolkit to the X11 distribution.

At the ITC, our experience with wide-scale software distribution has been a rocky one. Our releases to the CMU campus, to IBM (our primary source of funding), and to the X11 tape have all been preceded by sleepless nights spent building compilable systems and portable construction procedures. ("Hmm ... that code worked just fine for Mortimer when he checked it in ... wonder why it won't even compile now?") These releases have revealed problems with our *Make* scripts, our bug tracking mechanisms, and our source control mechanisms.

Only heroic effort and dedication have overcome these problems which can ruin the delicate environment in which researchers thrive. For example, talented programmers may be required to spend weeks on something as trivial and unsatisfying as preparing patch files.

We believe that the solution to such annoying and expensive problems lies in a discipline that we call **software management**. Software management touches broadly on diverse areas as software tools, database technology, and organizational issues.

In this paper, we begin by defining software management and the problems we assign to that area. We go on to describe a coherent set of policies and techniques called **hierarchical software management**. The

[†] UNIX is a trademark of Bell Laboratories.

bulk of the paper is spent describing a set of tools for software construction (a la *make*(1)) in a hierarchical development environment. (We should note before we lose too many more readers that the design described presents novel UNIX-flavoured solutions to perennially discussed *make* problems such as *make* script generation (or elimination), dependency tracking, parameterization of an installation and so forth. Those who are only interested in the technical stuff, and not so much the rational behind it, should turn at once to section 6.) The paper wraps up with a design sketch for a new source control scheme (which, unlike the construction system, has not been implemented yet), some observations about the need for flexible database technology, and of course, the ubiquitous "Conclusions" section.

2. What is Software Management

"... and why should I care? After all, the introduction said that this paper was about a research organization. Clearly the bozos at a research organization just can't hack the demands of real world software distribution. Here at Production Software Inc. our programmers know what it takes to get the job done."

Or so you might think. But software management was defined in [Tilbrook, Stern 87] as "the discipline of managing, maintaining, and distributing a body of software in source form". In other words, software management is almost every manipulation of source code other than its development by programmers. Source control transactions, *make* script generation, and "software shrink wrapping" are all software management activities.

Software management is not just a problem for researchers; it's a problem that comes up wherever there is an attempt at organized cooperative coding and distribution. It is not so much that our researchers don't know what "what it takes to get the job done"; they almost certainly do. Rather, it's that our research programmers are impatient when they perceive that, for their recurrent software management difficulties, "there ought to be a better way"

When we were first formed into the Software Management Team (or, affectionately, SMuT), we began by asking the development staff to tell us all of their software management problems. Most of what we heard fell into just a few categories, which are briefly considered below: controlled code sharing, automated software construction, software tracking, and software shrink wrapping

2.1. Controlled code sharing

Time after time at the ITC, one programmer or group has released new versions of libraries, header files or programs which break someone else's application. This is not a case of malice, or even carelessness. The intricacies of semantic dependencies between various pieces of code is complex and very hard to understand. (Locally, the situation is even worse because we use a dynamic linker which multiplies with surprising efficiency the number of ways version incompatibilities can arise).

Although we can dictate coding practices that reduce such problems, such as preserving library interfaces, or the set of symbols exported from some module – such coding practices are not always achievable. This is especially true in a cutting-edge research environment where the best designs for inter-modular interfaces become apparent only after a lot of code has been written to a bad interface.

We have recognized that incompatible changes are inevitable, and have sought ways to force human beings into the code release loop wherever incompatibilities arise. We design our tools to allow *people* to make the decisions in complex situations.

2.2. Automated software construction

Compiling and installing software is a complex, timely, and costly process. It is also critical. After all, a customer's first impression is often the ease – or difficulty – with which a product installs on his system. On another front, the validity of any testing done at the development sight relies upon the correct compilation and installation of the software being tested.

Conceptually, source is transformed into an installation by a long script of construction commands. Stu Feldman's *make* program injects that script with the notion of dependency and consistency in order to avoid redundant compilations. Unfortunately, using hand crafted *make* scripts to describe dependencies and construction scripts has proven to be tedious and errorprone.

A thorough *make* script which lists every dependency is unlikely to be portable because header files and libraries are in different locations on various target systems. An incomplete, though portable *make* script will eventually result in a botched/admirable/faulty installation.

Additionally, the weak parameterization mechanism offered by *make*, essentially just macros, has proven awkward when the body of source is quite large, and many *make* scripts are involved. *Make* does not scale well to even moderately large bodies of source.

We have extended the *make* concept into a powerful suite of tools which solve these problems. Our tools are described in the formerly alluded to technical section of this paper.

2.3. Software tracking

We have found that it is surprisingly hard for an individual to monitor the software on our system. Commonly used software management tools provide only marginal help for answering questions like, "what source files have changed since the last release to MIT?"

A first step towards solving this problem is to add *centralized transaction logging* to our source control system and some simple databases to help maintain the integrity of the source tree.

3. Shrink wrapping

Although we are predominately a research organization, we also find it important to distribute our software widely. We hope that our installation mechanisms will be powerful enough to adapt to a wide variety of systems and be easy enough for only moderately experienced people to use. Our goal is to be able to produce such a distribution easily and confidently.

Before we can meet these goals we must design construction mechanisms which are adaptable and are consistent across all of our software. We must have tools to extract exactly those source files which belong in a distribution: no more and certainly no less.

4. Understanding Software Management Problems

Once we understood the sorts of software management problems we had, we searched for their causes. We were fortunate to have the opportunity to study in detail a major release of software from the ITC development environment to the CMU campus. These releases typically required over-dedicated individuals to spend sleepless nights collecting necessary folklore and performing the installation.

We were not disappointed. The particular release we studied graciously presented us with every sort of problem people had experienced in the past. We came away with two observations:

Observation 1: There is a need for good software management tools.

Like most sites, the ITC has relied upon a source control system (RCS), *make*, and a plethora of shell scripts for all its software management needs. At critical times, these tools were not enough. Conceptually trivial tasks, such as generating a master source list, were difficult to carry out. Conceptually hard problems, such as multiple machine-type installations from a single set of source, were accomplished only by complicated kludges. Our construction system, for example, was becoming an unmanageably complex collection of shell scripts.

It seemed as though the investment necessary to write a new software management toolset would yield a substantial pay off. In sections 4 and 5 we discuss parts of our design for such a toolset.

Observation 2: Vital software management information was being lost in folklore

The ITC's programmers are partitioned into groups – each working on a different project. We observed that programmers suffered from few software management problems with the software from their own group. Problems showed up when software was passed from one group to another, or from several groups to some outsider (such as a CMU system administrator trying to install our software for the campus). For example, every group used subtly different *make* script conventions which, when the software was treated as a whole, made carrying out installations tricky.

In light of this observation, we sought ways increase the flow of software management information between groups. The result is described in the next section.

5. The Hierarchical Model of Software Management

Hierarchical software management is an organizational technique designed to solve the information flow problems described in the last section. The basic idea is quite simple: the various groups within an organization are structured into a tree hierarchy. Information flows between nodes of the tree the structure of which is reflected in the structure of the organization, and of the file system containing development source.

The root of the tree is a special repository for source, and for software management information. One person, called the *software gatekeeper* [Tilbrook, Stern 87], is placed at the root. The roles fulfilled by the gatekeeper are:

- the maintainer of the source database which we call the One True Source (OTS)
- the chief liaison between the groups which are the children in the tree, and between the development site and its outside clients
- the collection point not only of source, but of the software management information that goes along with it.

At each node of the tree which represents a development group, there is a *project gatekeeper*. The project gatekeepers' relationship to their group, and to the software gatekeeper, is analogous to the relationship between the various groups, the software gatekeeper, and the outside world. The project gatekeeper is a communication channel between their group and the software gatekeeper, and they are repositories of source and software management information that has not yet left their group.

Projects may contain subprojects, and the leaf nodes of the hierarchy are the programmers themselves.

We discovered that this hierarchy presents a rich paradigm in which to consider software management problems. For example, a natural bug report mechanism emerges: bug reports are given to the software gatekeeper who analyzes them and either responds directly (if the bug is known and has a known fix), or passes the bug down to the appropriate project gatekeeper. They in turn either responds or hands the report further down.

Just as bug reports flow from the root to the leaves, source itself flows from the leaves (the programmers) to the root. As code is passed up the tree, each gatekeeper can apply a set of criteria before accepting the source. The strictest criteria are those applied by the software gatekeeper at the root. The gatekeeper's source database, the OTS, represents the source closest to release. By carefully controlling what goes into that database, she is spreading the cost of predistribution clean-up over a larger part of the development cycle.

The gatekeepers are also in a position to prevent surprise occurrences of one group's code breaking another's. Roughly speaking, we specify that when developers want to compile against the "latest version" of some source file, they look first for their own private copy. If they do not have one, they look to the source being held by their project gatekeeper. Their query proceeds up the tree and if no other version is found they will by default use the version of the file that is in the OTS.

The upshot of this is that by accepting new code, project gatekeepers enable their own group to immediately begin sharing that code without interfering with the work of disinterested groups. Before that code is admitted into the OTS, the software gatekeeper has an opportunity to test for conflicts and, if necessary, arrange for their graceful resolution.

We admit to having introduced a rather unusual description of the "current version" of a source file. We are justified in this by the observation that without such structure the code sharing actually practiced is much more complicated and much less safe.

There are other benefits to hierarchical software management. For example, the software gatekeeper is in a position to guarantee that the test installation of the organization's software is always up to date with respect to the OTS. Another example: the hierarchy of acceptance criteria imposed by the gatekeepers traps simple mistakes early on and affords programmers the opportunity to correct them before they become expensive.

In order to fully realize these benefits, we need new software management tools. Better construction tools, source control tools, and software auditing tools can help realize the promise of hierarchical software management. We describe some of these tools below, beginning with a system that replaces *make*.

6. Automating Software Constructions

We had many goals for our construction system. In light of our holistic approach to software management, it had to coexist nicely with, and hopefully support, hierarchical software management. In addition, we wanted to fix all of the familiar problems with *make*: tracking dependencies, maintaining *make* scripts with an even moderately large number of source files, honoring the same set of conventions across many *make* scripts and parameterizing the construction process.

We also hold a deep respect for what it takes to write good UNIX tools. We wanted to write small, manageable programs which could be combined into a sophisticated construction system. We find fault with "monolithic" construction systems which try to cure all the world's ills with one massive hyperintelligent program. Such programs are costly, liable to break, inflexible, and fail to merge well with the rest of the UNIX programmer's toolset.

Key observations about *make* scripts

make scripts are effectively databases for two pieces of information. First, they contain the construction script – the list of shell commands necessary to compile and install a body of software. Second, they contain dependency assertions – an ordering relation on source files which can be used to avoid unnecessary or redundant compilation.

Dependency relationships are typically of a program upon its source, and the header files included in that source. Most dependencies are header file dependencies, and these can be extracted automatically by scanning source files for include directives.

Construction scripts can be described in a very high level fashion (e.g., "Build a library from all the .c files"). This high level description can be decomposed (e.g., "Compile the .c files into .o files. Archive the .o files into a library.") This decomposition can continue until the steps are quite simple. Although there is a lot of variation amongst construction scripts, they are all ultimately built up from simple rules. The paradigm of iterated decomposition suggests that traditional programming techniques can be successfully applied to the problem of script generation. Therefore, we have invented a tiny language for writing script generators. We have provided a library of moderately high level script generators for this language. Rather than writing *make* scripts, our programmers write small programs in this language. Because their programs can call on our library, they are generally extremely small (typically one or two lines).

With automatic dependency tracking and automatic script generation, it is possible to eliminate *make* scripts altogether. Many have tried to do just this to varying degrees. The *make* program used to build X11, and fourth generation *make* [Fowler 85] provide examples at the extremes of small and large scale *make* script reduction.

The problem is to generate *make* scripts accurately and cheaply while providing enough flexibility to perform the most esoteric construction. Dynamic dependency tracking is notoriously expensive and hard to do in a manner that is easily adaptable to new programming languages and preprocessors. Automatic script generation runs an even greater risk of being too inflexible to be useful. Below, we describe our tools and how they avoid these traps. Each tool takes care of a different part of the construction process. We follow our description of the tools with a description of the pipeline that puts them together.

Dynamic dependency tracking

Most other systems we know of perform dynamic dependency tracking only when the programmer takes some explicit action (such as typing "make depend"). We are not sure whether this is because dependency tracking is so expensive in these systems, or whether it's the other way round. We decided early on that the automatically generated dependencies should be updated with every construction. Our reasoning was that too many factors could change the dependencies (a change to a header path, the removal or addition of a new header file, a change to the library path ...). With so much going on, it is all but impossible for a programmer to decide when the dependencies need to be recalculated. In fact, a rough examination

suggested that it required at least as much computation to decide whether or not to redo dependency tracking as it would take to go ahead and redo it.

The tool we designed was *inclgen*. Inclgen accepts a list of source files, and produces a partial *make* script describing the dependencies of those source files on their header files. It uses regular expression rules (different rules for different source languages) to find include directives. It has no knowledge of conditional compilation, and so it can produce more dependencies than are truly there. This doesn't matter, so long as our *make* understands that certain header files are allowed to be missing. A sample *inclgen* invocation might look this way:

```
% inclgen sandalwood.c fileio.c screenio.c

sandalwood.c:S: sandalwood.h

fileio.c:S:      sandalwood.h\
               /usr/include/stdio.h

screenio.c:S: /usr/include/stdio.h\
              /usr/include/curses.h\
              /usr/include/sgtty.h\
              /usr/include/sys/ioctl.h\
              /usr/include/sys/ttychars.h\
              /usr/include/sys/ttydev.h
```

In the output, “:S:” is a notation to our *make* which means that these dependencies are of a source file upon its header files. Our *make* treats such dependencies with slightly different semantics than other dependencies because *inclgen* doesn't understand conditional includes. If *inclgen* is run on *mungdir.c* which reads:

```
#if BSD /* use the systems dir.h if it's present */
#   include <sys/dir.h>
#else
#   include <ourhdrs/dir.h>
#endif
/* include a header file that will be built during the construction */
#include    "lexoutput.h"
```

it will produce (on a BSD system) the dependencies:

```
mungdir.c:S: /usr/include/sys/dir.h\
             /usr/andrew/ourhdrs/dir.h\
             lexoutput.h
```

On a non-BSD system, where */usr/include/sys/dir.h* does not exist, it will produce:

```
mungdir.c:S: sys/dir.h\
             /usr/andrew/ourhdrs/dir.h\
             lexoutput.h
```

When either *make* script is run, our version of *make* will try to make *lexoutput.h*. If the second *make* script is run, our *make* will also try to make */sys/dir.h*. Of course, there are no rules for making that, but the “:S:” tells *make* that these dependencies were generated without regard for conditional includes. Rather than exit, our *make* will just say:

Warning: Didn't have a rule to make “sys/dir.h”.

If *sys/dir.h* was really supposed to be there, the recipe associated with *mungdir* will fail, and the construction will stop.

Inclgen achieves speed by caching as much information as it can in a file in each directory in which it is invoked. A user of our construction software will find the file `Vincls-Cache._` in each of his source directories. The cache saves inclgen the work of having to scan source for include directives each time it is invoked. In the example shown, the cache file might begin:

```
sandalwood.c 577322688
^#[ ]*include[ ]*\([<"\)\([>"]*\)
\\2
<stdio.h
```

We'll leave that unexplained. Interested readers are invited to contact one of the authors for more information.

Automatic construction script generation

Automatic script generation is an almost paradoxical problem. On the one hand, there is the sense that very little information is needed to describe even very long scripts. "Turn each .c file into an executable program installed in \$BinDir," is all a programmer should have to write. On the other hand, hard-wired script generators fail to cope with new languages or preprocessors, and generally lack the flexibility for real-world construction problems.

Our solution is to replace hard-wired script generators with a tiny language called *pmplus*. Rather than *make* scripts, we write *pmplus* programs to generate *make* scripts.

The *pmplus* language is built with no inherent knowledge about construction scripts. Instead, it has good primitives for manipulating file lists, support for shared subroutine libraries and a parameter passing mechanism that resembles traditional command line switches.

We provide a rich library of subroutines for script generation. Typically, to replace a *make* script a programmer can write a one line *pmplus* program that invokes the appropriate subroutine with the right arguments. For more esoteric constructions, the full programming power of the language is available.

Scripts generated by the standard library all follow the same *make* script conventions for parameterization and the naming of virtual targets (such as Install, or Clean). What is more, the library scripts are free to generate slightly different recipes for different target environments (Should you ship a *make* script that runs ranlib?).

A full description of the *pmplus* language would fill this paper. Therefore, we will leave you three examples of its use. The first is a *pmplus* program that was used to replace a 20K *make* script. Incidentally, it produced a 90K *make* script which indicates that there were a lot of missing dependencies in the original *make* script.

```
; This pmplus program invokes the library subroutine on its
; argument files.
; It tells library to produce a library named libmessageserver.a,
; which will be installed in the default location.
library -n libmessageserver.a @argv
```

The second example is the subroutine *library* which was invoked by the last example:

```
; This summary section describes the use of this subroutine.
; It is also parsed by the interpreter to do automatic argument cracking:
summary:
```

```
Make objects from the argument files and archive them.
```

```
flags:
```

```
-L      Local only. Don't install the library.
```

```
        This is assumed if no name is specified for the lib.
```

```
keywords:
```

```
-n name  Name the library.
```

```
-d dir   install the library in 'dir' (LibDir is the default)
```

```
endsummary
```

```
; First, decide on a name for the library.
```

```
; We look at how many "-n" arguments were provided.
```

```
if @nkeyword 1
```

```
    fatal: ESCHIZO: too many names for this library: @nkeyword
```

```
elseif @nkeyword 1
```

```
    ; these are assignment statements.
```

```
    = name @nkeyword
```

```
    = namearg -n @nkeyword
```

```
endif
```

```
; the "mklib" subroutine is a lower level call
```

```
; that knows how to make the library, but no install it.
```

```
mklib @namearg @argv
```

```
; in this script, we add make rules to install it.
```

```
; If no name was provided (nkeyword has no value)
```

```
if @nkeyword 0
```

```
    ; then pretend we were called with -L
```

```
    ; (Local only ... no installation)
```

```
    = Lflag 1
```

```
elseif !@Lflag && @dkeyword 0
```

```
    ;else if we're not -L, but no destination was offered,
```

```
    ;use the default.
```

```
    ; By convention, "LibDir" will be replaced by the
```

```
    ; default library installation
```

```
    ; directory in our makefiles.
```

```
    = dkeyword LibDir
```

```
endif
```

```
; now that we know more about what we've been asked to do, make sure
```

```
; it makes sense.
```

```
if @Lflag && @dkeyword
```

```
    fatal: -d requires a -n and is incompatible with -L
```

```
endif
```

```
; if it does, then for every destination specified, call the
```

```
; "install" subroutine to generate the rules which will install
```

```
; the library
```

```
for destination in @dkeyword
```

```
    install -r -R -d @destination @name
```

```
endfor
```

The final example is of an invocation of the *pmp* interpreter. The *summary* section which is parsed for argument *cracking* is also available for on-line help. Since we depend upon a large library of subroutines, we decided to make it easy to get help.

```
% pmp -f library - -x
Make objects from the argument files and archive them.

-L      Local only. Don't install the library.
        This is assumed if no name is specified for the lib.
-n name  Name the library.
-d dir   install the library in 'dir' (LibDir is the default)
```

Automatic source identification

There are several reasons why we would like to identify source files mechanically. For example, it is convenient if the filename arguments to a script generator are provided automatically, rather than requiring the programmer to edit a file every time he creates or destroys a source file. Another example: we would like to perform software audits that produce separate lists source files, source control files, objects, binaries, and junk files. Identifying source files automatically makes this possible.

To enable source identification, we institutionalize conventions which are already present less formally. We specify a (user-extensible) list of extensions and require that every source file ends with one of those extensions. In some cases, this is not enough. Parser.y (a yacc file) might produce *parser.c*, which should not be mistaken for source. To prevent such a mistake, our list of extensions is partially ordered. The extension .y is greater than the extension .c, which is interpreted to mean that if *parser.y* and *parser.c* are both present, *parser.y* is a source file and *parser.c* is not.

Our rules are applied by a small filter called *srcxtrct* (source extract). For example:

```
% echo parser.y parser.c parser.o blather noho.sh | srcxtrct
parser.y
noho.sh
```

Specifying parameters

There are many places to parameterize construction scripts. The search paths for headers and libraries, flags to compilers, the names of target directories and the default flags to an install program are all examples. Often parameterization is incomplete and awkward. A user may have to edit many *make* scripts to establish the default values for parameters and override those parameters where necessary.

There are three aspects to our solution of parameterization problems. First, we make sure that the script generation code in our *pmp* library applies the same parameterization conventions across all *make* scripts generated. Second, we use a cpp-like preprocessor called *mpp* (differing from *cpp* only in some built-in macro, and in the comment syntax) to expand the parameter macros imbedded in our scripts. Third, we provide the user with a mechanism to specify the default value of parameter macros in a single file called *LclVars* [Tilbrook, Place 86].

A *LclVars* file contains assignment statements of the form:

```
set  macro      value
```

When *mpp* is run on a construction script, it replaces all occurrences of *macro* with *value*. The bindings in a *LclVars* file are inherited in subdirectories of the directory containing it. Therefore, it is possible to bind some script parameter for an entire source tree by modifying the *LclVars* file at its root. *LclVars* files in subdirectories may augment or override the inherited values.

Mpp is not the only program that uses *LclVars* variables. *inclgen*, for example, finds header paths by scanning *LclVars*. *srcxtrct*'s suffix rules can be modified by *LclVars* variables. Controlling all of these programs with a single mechanism simplifies constructions greatly.

Source paths

Hierarchical software management involves code sharing, as described in section 3. Recall that when a developer wants the current version of a source file he must look at each gatekeeper node between him and the root. His query will end, if nowhere else, at the root of the tree and the One True Source.

We chose to implement these code-sharing semantics in a very literal sense. We maintain one source tree called OTS. Every project gatekeeper has their own source tree which contains any source files shared by their project, but not yet released to the OTS. Finally, developers keep their own trees with whatever source they are working on.

The conventional software management technique for code sharing involves either copying all shared files to each developers directory or creating symbolic links from the developers directory to a master source tree. Copying the files invites divergence problems when the master source database is updated, but developers trees lag behind. Symbolic links are hard to maintain in a sure-fire way.

Our favorite source-sharing technique resembles the VPATH mechanism in [Fowler 85]. Roughly, the construction tools search a path (called the source path), for every source file needed. A path mechanism prevents the divergence that can arise from copying source files, and is more manageable than symbolic links.

In a hierarchical environment, the OTS tree is the last element on every source path. A developer's private source tree is the first element, and various project gatekeeper's trees are between those two. Conceptually, each of the project trees acts as a filter on the OTS – preventing old versions of files from passing through by overriding them with new versions.

Each directory has a source path which is defined as a *LclVars* variable. A *LclVars* file in a source tree (*~user/LclVars*) might read:

```
set srcpath ./cmu/itc/smtree/@subdir /cmu/itc/ots/@subdir
```

As *LclVars* values are inherited to subdirectories of the directory in which they are defined, the symbol *@subdir* expands to the relative path from the *LclVars* file to the subdirectory. So in the directory, *~user/src/inclgen*, the source path is:

```
./cmu/itc/smtree/inclgen /cmu/itc/ots/inclgen
```

An *ls*-like program called *sls* lists files along the source path rather than just those in the current directory. Only the first occurrence of each file is shown and so *sls* output is effectively a list of the latest versions of each source file.

A new make program (almost)

Our software construction system generates and then interprets a *make* script. We might have just used existent *makes* as our back end, but there were three problems: First, no two *makes* are alike. An infuriating amount of attention must be paid to the subtle semantic differences between them. Second, *make* scripts offered more than we needed. Since we have a script generator to produce our recipes, suffix rules are redundant. Since we have *mpp* to preprocess our *make* scripts, *make*'s macros were redundant. Third, *make* didn't do enough for us. We believe that a good *make* should record the recipe and dependencies used to build every target. Regardless of time stamps, a target should not be considered up-to-date if either changes. We wanted a *make* that would recompile C source if we changed the compiler flags from *-O* to *-g*.

We surveyed the set of existent *makes*, and though we found none that was directly suitable, we were attracted to the minimalist style and clean semantics of Andrew Hume's *mk* [Hume 87]. We emulated these and wrote *mimk* (pronounced mimic). *Mimk* extends *mk*'s use of attributes to precisely control the semantics of each rule. (The “:S:” mentioned while describing *inclgen* is a compound attribute; it applies three simpler attributes to the rule.) *Mimk* also implements the changing recipe semantics we wanted. Finally, *mimk* eliminates what we don't need: macros and default recipes.

The construction pipeline

Many programs are involved in our construction system. It's no longer just a case of running *make*. Our tools fit together this way:

We begin by identifying the list of source files. This list is stored in a file called *SourceList._*, taking care to make sure that the timestamp of *SourceList._* only changes when necessary.

```
sls | srxtrct | putifdif ./SourceList._
```

Rather than a *make* script, programmers write a *pmplus* script stored in the file named *PMCS*. We cache the output of interpreting *PMCS* and only reinterpret the script if either the *PMCS* file, or the source list is newer than the cached output (called *Pmak._*). If we must reinterpret the script, we use:

```
pmp -f PMCS - 'cat SourceList._' > Pmak._
```

The construction script is preprocessed with every construction.

```
mpp < Pmak._ > Mkfile._
```

Dynamic dependency tracking takes place every time as well.

```
inclgen < SourceList._ >> Mkfile._
```

And finally, we interpret the *make* script.

```
mimk -f Mkfile._ targetname
```

We make this complexity manageable by use of a front end. The program *pmak* understands the intricacies of this pipeline, and presents the user with an interface no more complicated than conventional *makes*.

Costs and Benefits

It should be obvious that our system will always be slower than conventional *makes*. Simply put, this is because it does more. We find it irrational to remove functionality for a few seconds off our make times because (1) the actual compilations are responsible for one or two orders of magnitude more time than our construction system and (2) the few extra seconds yield correct *make* scripts with consistent conventions.

7. Source Control for Hierarchical Software Management

An important part of hierarchical software management is correctly maintaining the OTS tree and the individual project trees. We have designed new semantics for source control mechanisms to help with the process. These semantics have not been implemented yet, so we will be brief in our description of them. A good deal of our design comes from discussions with Adam Stoller, one of our project gatekeepers, with whom the check-in semantics described originated.

The first novel characteristic of our design is the lock semantics. In conventional systems, a version is either locked or unlocked. In our system, the lock on a version travels up and down the tree. When the lock is held at some node (such as a project gatekeeper's node), the corresponding version can only be locked by descendants of that node (members of the project group, for example). Initially, all locks are at the root. When a developer releases the lock on a file, the lock does not return to the root, but to that developer's project gatekeeper. As the new version passes acceptance criteria, the gatekeeper passes the lock further along until it returns to the root.

The second novel characteristic of our design is the motion of a source file as it is "checked in". When a developer checks in a source file, the lock is handed to his project gatekeeper, the source is deleted from his own tree, and copied into the project tree. In this way, the code becomes shared immediately upon check in. We provide a mechanism by which gatekeepers at critical nodes can refuse a lock (and its attendant source) until they are convinced it is safe to begin sharing the code.

Finally, our source control system is designed to log every transaction in a central file that is part of the OTS tree. The information in that file helps the software gatekeeper to keep track of what parts of the source are changing and where her attention is needed.

8. Future Directions, Database Technology

As our software systems grow, software management problems grow with them. We foresee a time when the amount of information needed to understand the status of a system will become unmanageably large unless new tools are brought to bear. For example, we would like to be able to extract from our source control database the exact version of our code that was shipped to some site. We must be able to easily generate a version list for that site. Given that version list, we must easily be able to extract the appropriate source files – even though some files may have changed names or may no longer be part of the system at all. Another example concerns bug tracking. It is handy to know what bugs are fixed in what versions of the code. It saves time if the software gatekeeper can consult a database of known fixes before bothering developers.

The short of this is that there is a complex web of various relationships woven over the development site source. We believe that these relationships will become obstacles unless explicit database technology is used to represent them. This represents an area of future research.

9. Conclusions

We have discussed a wide range of issues in this paper, and so it seems reasonable to pick out a few points to sum things up:

- 1) Software management is a hard problem. Make and RCS only begin to address some of the issues involved.
- 2) A project structured development environment may suffer from weak communication between the groups. Hierarchical software management can be effective in enhancing communication.
- 3) Good tools are also necessary for software management. We have discussed a make replacement in detail, and a source control system in overview.
- 4) We believe that database technology is the next logical step for software management research.

10. Acknowledgment

The inspiration for this work and many key ideas came from David Tilbrook. The architecture of the construction system grew out of his previous work ([Tilbrook, Place 86]). He supervised and contributed much to the design process which would never have happened without him. Linda Branagan provided invaluable help preparing this paper.

References

- [Fowler 85] *The Fourth Generation Make* Glenn S. Fowler, Usenix Portland Conference Proceedings Summer 1985
- [Hume 87] *Mk: A Successor to Make* Andrew Hume, Usenix Phoenix Conference Proceedings Summer 1987
- [Tilbrook, Place 86] *Tools for the Maintenance and Installation of a Large Software Distribution* D.M.Tilbrook and P.R.H.Place, EUUG Florence Conference Proceedings, April 1986
- [Tilbrook, Stern 87] *Cleaning up UNIX Source - or - Bringing Discipline to Anarchy* David Tilbrook and Zalman Stern, EUUG Dublin Conference Proceedings September 1987

Retaining SUID Programs in a Secure UNIX

Steven M. Kramer

SECUREWARE, Inc.

Suite N-108, 430 10th St., Atlanta, GA 30318 (404) 894-5170

ABSTRACT

This paper shows the *set-userid* (SUID) program to be insufficient by itself to promote a secure environment. Various protection mechanisms such as auditing, identity stamps, privilege sets and promains can alleviate the user's security concerns regarding the SUID program. A new protection concept for the UNIX Operating System is defined. Called a *promain*, or *protected domain*, it allows SUID application migration from insecure systems to secure systems. Promains are primarily intended as a security tool for the average user, yet they may be applied to system management tasks as well.

1. Introduction

This paper presents a new protection model for the UNIX¹ Operating System. The new protection model serves to enhance protection for the users of the system and works in concert with, not instead of, all the current protection mechanisms. Most other protection mechanisms in UNIX are targeted for the protection of the system against the user. There is a strong need to provide features that the user can use if he chooses to protect his data from misuse by others.

A new feature is discussed here that allows the retention of *set-userid* (SUID) programs while providing privilege attenuation at the same time. Keeping SUID programs in UNIX is desirable because there is a great deal of dependence on the selective parceling of authority that a SUID program provides. The retention of the SUID mechanism is important to allow application portability to a secure environment, and to provide for controlled sharing of resources without granting total privilege.

Secure UNIX implementations that remove or replace the SUID program in trying to meet criteria in [3] sacrifice application portability and protection unnecessarily. One such implementation is proposed in [8]. The proposed solutions are generally cumbersome and hard to comprehend by a typical user or administrator. While they offer privilege attenuation, the mechanism itself is difficult to model or maintain.

While the features described are targeted for System V environment, they are also meant to be exploited in other environments, including POSIX [7] and Berkeley-based systems, with minimal effort.

The remainder of this paper is organized into four sections. Section 2 reviews the existing protection mechanisms of the UNIX operating system. Section 3 explains privilege enhancement and resulting security problems. Section 4 details a protection model that specifically addresses those problems, including examples of the use of the model. Section 5 presents some conclusions.

2. Classic UNIX Protection Features

The classic protection features of the UNIX Operating System have been widely detailed in the literature of the past decade ([1], [5], [10], [13]). The SUID mechanism is given exhaustive attention in [1]. This section serves to review the traditional UNIX security mechanisms. The features used in the

1. UNIX is a registered trademark of AT&T in the USA and other countries.

UNIX Operating System include: users, groups, file protection, IPC protection, SUID programs, and *chroot*. The resources of concern for security in the UNIX Operating System are processes, files, and IPC entities.

The following list reviews the important security features of UNIX:

Users, Groups and Processes

The UNIX Operating System recognizes different users by virtue of unique user IDs, or UIDs. Each resource in the system is assigned an owner so that a resource can be easily traced back to a distinct user. A similar attribute is the group. Each group has a unique group ID, or GID. Several users may belong to the same group, and the group is part of the determination of access rights.

Processes are viewed as the active resources in the system and each process is labeled with the user to which it belongs. In most cases a process is identified with one user, but in SUID programs it is labeled with two users – the real user running the process, and the effective (temporary) user to which the process is assigned for a particular task. Like the user identity, a process normally has one group associated with it but can have a separate real and effective group when running a *set-groupid* (SGID) program.

Processes are bound to a unique user and group at login time. Processes inherit the user and group IDs of their parents, and may be altered by SUID programs, SGID programs, or the *setuid(2)* system call.

Files

Files are the main type of passive resource in the UNIX Operating System. Like a process, a file is labeled with a user and group. It also contains a set of three permission fields: one for the owner of the file, one for members of the file's group, and one for all others. When the system mediates access between users and files, it uses the effective user and group of the process.

Most files must appear in the hierarchical UNIX file system. Pipes are the one exception discussed below. The access rights to a file are actually determined by the permission of the file itself and the permissions of the directories leading to the file. This distinction is important and crucial in all efforts to set up a hardened or secure UNIX system. The directory path(s) leading to a file can be used as a firewall to users in the system.

IPC Entities

The System V IPC entities exist in a name space outside the file system and share identical protection mechanisms. The mechanism is closely related to the user/group/other permission model for files, but lack the hierarchical ordering of objects.

The Superuser

The superuser has an effective UID of 0 and typically a login name of *root*. The superuser is the only user with privilege to perform certain system services. Some examples of the superuser privilege are mounting file systems on an existing directory, setting higher priorities on processes, locking data in memory, raising limits and creating special files. The superuser, by virtue of its unique UID, bypasses the conventional access checks for files and other objects.

Along with the superuser power must come control. The control of the distribution and use of the superuser authority is the topic of other papers, and although a primary concern of secure UNIX developers, is beyond the scope of this paper. See references [5], [6], [7] and [13] for more information on this subject.

chroot

The *chroot* facility is invoked by the superuser to limit access to the file system by a process. A subtree of the current hierarchical file system is designated and becomes the new root ('/')

directory. This action is irreversible.

The SUID Program

One of the most interesting and least understood security mechanisms of the UNIX Operating System is the patented SUID program ^[11]. The SUID program allows user A to run a program owned by user B and retain the effective rights of user B. The rights of user B are retained until either the process exits, or the program issues the *setuid*(2) system call with an argument equal to the UID of user A. In the latter case, the rights of user B can be reasserted by issuing the system call *setuid* with the argument equal to the UID of user B. This inversion of the effective user can continue until the process exits or invokes another program as user A.

SUID programs are used for two types of protection. The first is a SUID program which allows a general user to perform a privileged system task for a short duration. The *mount*(1) program is an example. The other use of SUID programs is where a SUID program is owned by a user that is not the superuser. A representative SUID program of the second variety is used to gain controlled access to resources of another user that are not directly available to the invoking user.

There are some rules that the writer of a SUID program should follow to help ensure that the owner of the SUID program will not suffer ill-effects from misuse or abuse of the program. A detailed explanation is provided in ^[13].

It is worth noting that all the rules for SUID programs are one-sided. They serve to protect the owner of the SUID program against the invoker. When a SUID program is owned by benign users, such as system personnel, the rules make perfect sense.

SUID programs may also be written by a general user to control access of his private resources to others. The rules above serve to protect his SUID programs as well as they do for system personnel. The SUID mechanism as it now exists in UNIX may actually increase the varieties of Trojan horses when it can no longer be assumed that the SUID program owner is benign. This problem is reflected in both the above rules and the existing UNIX protection model, and is discussed in detail in the next section. A mechanism to control the problem is discussed in section 4.

3. Problems with Privilege Enhancement

This section describes the problems of the SUID mechanism. It demonstrates the traditional SUID mechanism as a form of privilege enhancement, and suggests some security problems. Appendix 1 contains an example of SUID privilege abuse.

The SUID mechanism of the UNIX Operating System generally to limit the privilege of an invoking user to a system, subsystem or user-controlled resource. Assuming proper programming practices, the owner of the SUID program can be assured that the invoking user cannot manipulate the underlying resource improperly.

The SUID mechanism does not, however, address the inverse protection, namely the protection of the invoking user from misdeeds by the SUID program owner or creator. Because a SUID program is really a convergence of two users' privilege, either user can theoretically gain control. Appendix 1 shows a case where the SUID program owner usurps the authority of the invoking user. The rules in writing a SUID program protect the owner of the SUID program; there is no current mechanism to protect the invoker.

The invoking user of any program is always subject to hidden, malicious code called a *Trojan horse*. A Trojan horse that additionally copies itself called a *virus*. Without source code and translator examination tools, any program could perform hidden actions. A user largely depends on the site administrators to review and/or test all system programs and certify them safe for general use. Other

programs for which the user can and does perform a thorough code review are also generally safe.

The other classes of programs are those that are imported into the system from outside the control of the administration, and those created from other users of the system. These two classes of programs both contain the same high risks.

In System V, a SUID program can alternate freely between the SUID program owner and the real user, each in turn becoming the effective user of the program. Since the program can switch between the two users so easily, it has access to all resources and privileges accessible to either user.

The invoking user of a SUID program has no control and often no knowledge of the semantics of the SUID program. The SUID program, besides providing access to objects of the program owner that are normally hidden to the invoking user, can also read, copy, or delete objects of the *invoking* user that are normally hidden to the program owner. Thus, there is an imbalance in the way SUID programs operate. The protection the SUID owner has in the algorithms of the program do not have an analog with respect to protection of the invoker.

No mechanism in the UNIX Operating System today can alleviate this problem short of the elimination of the SUID program itself. UNIX variants that do just this have done so at the expense of UNIX compatibility. The SUID feature is integral to the architecture of many application systems, among them: database management systems, transaction processing systems, office automation systems, user interfaces and network facilities. Clearly, a new feature is needed to assist the invoking user in the secure use of SUID programs.

4. SUID Attenuation

SECUREWARE has developed a three-pronged approach to handling the problems present with the SUID program mechanism without eliminating it or to sacrificing compatibility. The four mechanisms are: auditing, the *login* user, process privileges, and promains.

4.1 Auditing

Auditing, the precise recording of all security-relevant events in the system, is not a new mechanism, but it is new in the UNIX Operating System. Vendors are just beginning to include integrated auditing capabilities in their systems, beyond the simple and distinct mechanisms used for *su* and *login* activity. To be effective, auditing must handle *all* security events. It must also have a simple interface for site personnel and be well-protected to maintain its own integrity. An effective audit mechanism should not consume excessive disk space or degrade system performance in order to collect meaningful audit data.

The SECUREWARE Nonintrusive Extensible Auditing Kernel (SNEAK) package meets all of the above criteria. SNEAK has two features that directly aid in the monitoring of SUID programs. The first is that the auditing package monitors all successful and unsuccessful attempts to obtain resources, change the effective user ID (including both invocations of SUID programs and *setuid* system calls), and many other categories of system events.

In addition, SNEAK has the ability to limit the types of information in the audit trail (the human-readable sequence of events recorded by the auditing mechanism). SNEAK allows the site administrator to select the exact types of events, users and groups that are of interest. The administrator can concentrate on the SUID programs of suspected users. It is also easy to monitor events associated with SUID programs because other events can be eliminated from the audit trail. The events selectively collected in the audit trail can focus on errant SUID program behavior. The administrator can quickly determine if a seemingly benign SUID program is accessing private files.

Data collection with SNEAK is not available to general users and occurs without their knowledge or involvement. SNEAK, then, is able to get a true reading of system activity. SUID programs cannot determine if SNEAK is recording them and adjust their actions accordingly.

4.2 Login User

The login user is an attribute which is indelibly stamped on a process at the time of login, *cron* or *at* job startup. The login userid is an inherited process attribute. Once a login user stamp is set, not even the superuser can alter it by attempting to reset it. This provides an immutable identifier for both SNEAK and other system services that need to know the true user accountable for process actions.

The reason the login user concept is required is that through a succession of SUID programs and *setuid* system calls (with superuser privilege), both the real and effective user IDs can change. Once both are changed in a traditional system, there is no way to trace the true, invoking user of the process.

Rather than alter the well-known semantics of the real and effective user IDs, the login user ID is a complementary mechanism that preserves the old semantics while providing accurate process accountability.

4.3 Privileges

SNEAK is a powerful monitoring tool that reports on system activity, but it cannot prevent unwanted actions from occurring. SECUREWARE has adopted a protection feature that complements the existing features described in Section 2 to limit the rights of a process. Each process has an attribute called a *privilege set* which represents the set of actions that a process may perform. Other secure UNIX implementations have implemented privilege sets, each with a slightly different policy regarding their use ([2], [4], [8]). In SECUREWARE's implementation, the privilege set of a process is non-increasing, i.e., processes can remove privileges from the set but can never add privileges. The privilege set is inherited when the process performs a *fork* call.

Viewing this another way, given the initial privilege set P , where:

$$P = \{P_1, P_2, P_3, \dots, P_n\}$$

a new privilege set P^* may be derived as:

$$P^* = \{P_1, P_2, P_3\}$$

or

$$P^* = \{ \}$$

but not as

$$P^* = \{P_1, P_2, P_{n+1}\}$$

The last set is unacceptable because it is not a subset of P .

The actual privileges in the set are partially determined by the type of secure system product that is being run. For example, secure systems that handle terminal labeling² have a privilege for labeling. Once a terminal is labeled and the privilege is excluded, the label cannot be altered by the program. Some privileges are reserved for system personnel or system process and never appear in the privilege

2. A *terminal label* is a human readable marking on a terminal that identifies the system, the security level of the user, or some other identification mark. Such a label can only be placed or altered by the system, not by the user. Typically, a terminal label appears on the label or status line of a terminal.

set of a user. Others are meant exclusively for users. Except in a few cases, these are available to the user at login time and the user administers the set (by removing those not currently needed).

In determining permissions, the system looks at the privilege(s) in the process in addition to the traditional security controls. Access can be denied by either the lack of a privilege or by the traditional controls. If a user has all the user privileges, his processes will operate as they now do in a traditional implementation of the UNIX Operating System. If, however, the user employs the privilege set feature he will have a opportunity to control processes.

The new SECUREWARE program *privs* lets the user employ or change privileges. In one form, the user can execute one command with reduced privileges. For those cases where the user wants part of a session to be operated with altered privileges, he can create a subshell with *privs* and operate there, safely assured that the environment is more controlled.

4.4 Promains

The privilege that attenuates the power normally exercised by a program is the *promain*, or *protected domain*. The *promain* privilege, when set, does nothing to prevent SUID programs from operating exactly as before. When the user removes the privilege for a process, however, SUID programs that run in the process or any of its children are subject to additional constraints.³

Once the *promain* privilege is removed, SUID programs operate differently. Upon execution of a SUID program, a *promain* is *created*. When a SUID program is operating with the effective user set to the SUID owner, the access mediation is unchanged. When the SUID program operates with the effective user set to the real user (i.e., the program performs

`setuid(getuid())`

during the program execution), the *promain* that was previously created is made *active*. When a *promain* is active, the current directory becomes the *promain root*. All files in the subtree below the *promain root* are excluded from the extra checking of the *promain*. Access to all other resources, including files referenced outside the subtree and all IPC objects, depends on both the real user ID and the SUID program owner. If both users could not individually access, create, modify or delete a resource, access is denied in the *promain*.

A *promain* is extended when other programs are executed from the SUID program causing the *promain*. In this way, a *promain* cannot be removed by a program by executing another program to perform the actual damage. Similarly, once a *promain* is created, all children are born in the *promain* and are restricted by the same rules that apply to the parent.

The current directory at the time of *promain* creation is a process attribute that is retained through *chdir* operations. It is fixed at the time that the first SUID program is run. File links may not occur between existing files outside the *promain root* to files within the *promain root*, in order to prevent hierarchy changes from changing the protection afforded to a file. All other link operations are valid.

4.5 Applications of Promains

A user that creates the conditions for a *promain* by removing the *promain* privilege from his privilege set will be able to eliminate some of the potential damage of a Trojan horse SUID program. The *promain* will let all SUID programs run unaffected as long as the effective owner is the SUID program owner. When, however, the program changes identity to that of the invoking user, the system becomes

3. SUID programs owned by the invoking user are exempt from the additional restrictions and are treated as regular programs.

the invoking user's surrogate protector. It makes sure that, even though the program's identity is that of the invoking user, the program does not wield the full power provided to the invoking user.

The following chart demonstrates the effects of a promain on file access capability. Note the different behavior of files above and below the current directory.

Environment

Current directory - /ufs/smk

Invoking user - smk

SUID program owner - daa

Pathname	Owner	Mode	Effective User	Operation	Allowed?
/ufs/daa	daa	drwx-----	daa	chdir	Yes
/ufs/daa	daa	drwx-----	smk	chdir	Yes
/tmp/daa/see	daa	-rw-rw-rw-	daa	open	Yes
/tmp/daa/see	daa	-rw-rw-rw-	smk	open	Yes
/tmp/daa/see	daa	-rw-rw-rw-	daa	chmod	Yes
/tmp/daa/see	daa	-rw-rw-rw-	smk	chmod	No
/tmp/smk	smk	-rw-----	daa	open	No
/tmp/smk	smk	-rw-----	smk	open	No
/tmp/smk	smk	-rw-----	daa	chmod	No
/tmp/smk	smk	-rw-----	smk	chmod	No
/tmp/smk	smk	-rw-rw-rw-	daa	chmod	No
/tmp/smk	smk	-rw-rw-rw-	smk	chmod	Yes
/ufs/smk/tst	smk	-rw-----	smk	open	Yes
/ufs/smk/tst	smk	-rw-----	daa	open	No
/ufs/smk/tst	smk	-rw-----	smk	chmod	Yes
/ufs/smk/tst	smk	-rw-----	daa	chmod	No
/ufs/smk/free	smk	-rw-rw-rw-	daa	open	Yes
/ufs/smk/free	smk	-rw-rw-rw-	smk	open	Yes

The concept of a promain root allows a user to set up a controlled environment in which to execute programs. The user can widen access to files in the subtree, but allow access above the promain root to truly sharable files. The user is assured by the promain mechanism that although the SUID program can freely handle information within the promain, it cannot do so with data above the promain root. If the private data residing in a promain is compromised, not only will the user know the suspect, but he will also know that the damage done by the SUID program is limited to data in the directory.

As an example, assume user *smk* runs the SUID program *tricky* owned by *daa*. (An example *tricky* program that would cause the behavior below is found in Appendix 2.) He is worried that the program may try to usurp his privileges, so he chooses to use a promain. The following is a session segment showing the use of a promain and later the same program without using a promain:

```

$ prvs                # list the current privileges
suid,promain
$ prvs -r promain     # remove the promain privilege in a subshell
$ prvs                # list the privileges in the new subshell
suid
$ mkdir okdir
$ cd okdir
$ cp $HOME/protfile . # move needed files to here
$ ls -l protfile /usr/smk/hide
-rw----- 1 smk  sware   646 Jan 14 19:05 protfile
-rw----- 1 smk  sware   646 Jan 16 11:52 /usr/smk/hide
$ tricky
/usr/smk/hide: Permission denied # caught in the act!
$ exit                # leave the promain
$ prvs                # back into the original shell
suid,promain
$ tricky
$
$                      # this time the program worked
$

```

Notice that the malicious behavior of the program was thwarted in the promain but was allowed to perform unimpeded without an active promain. Promains are meant to handle exactly this kind of errant SUID program. Also notice that "protfile" was meant to be accessed since it was explicitly moved into the promain root.

A SUID program, like any untrusted program, can cause damage by executing hidden code (the Trojan horse). However, a SUID program can do no more harm than a regular program when running in a promain. In fact, it will do less harm because of the restrictive access mediation enforced in promains.

5. Conclusions

SNEAK, the login UID, privilege sets, and promains presented in this paper are simple ways to alleviate the problems associated with SUID programs. SNEAK performs a complete and non-intrusive monitoring function that can be tailored to the specific detection of SUID activity. The login UID assures that the human user at the terminal is known throughout the session. Privilege sets provide the user with a way to control the authority of users and programs. With the promain, users have a means for both testing and controlling the impact of foreign SUID programs. The accentuation of privileges and authority that plague the SUID program in traditional UNIX systems can be eliminated as a concern in systems that have promains.

Promains do not address similar problems associated with SGID programs. The reason is that, in the System V.2 base used, SGID programs do not have the identity cross-over problems of SUID programs have, i.e., SGID programs cannot switch identity back to the invoking group. Since the cross-over is the point where problems develop, SGID programs do not have the exact set of security concerns as SUID programs. For System V.3 and later systems, promains need to encompass SGID programs as well.

More research needs to be done to find additional ways to eliminate Trojan horses and viruses. Promains eliminate a class of Trojan horses from SUID programs but do not address Trojan horses in ordinary programs. A full or partial solution to Trojan horses, if any realistic one can be found, must also address UNIX compatibility in a similar manner to promains.

Appendix 1 - Example Trojan Horse in a SUID Program

The following short program demonstrates the dual identity that SUID programs can attain and how easily it is manipulated. Given current UNIX protection mechanisms, there is no way to prevent the program from performing what it does without reverting it into a normal program.

```
#include <sys/types.h>
#include <sys/dir.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    int dir_fd, in_fd, out_fd, amt, saved_uid, real_uid;
    struct direct buf;

    /*
     * The SUID program owner is careful about
     * his own files!
     */
    (void) umask(~0700);
    saved_uid = geteuid();
    real_uid = getuid();

    /*
     * Become the invoking user just in case the directory
     * is protected. The invoking user wouldn't like this
     * If he knew.
     */
    (void) setuid(real_uid);
    if ((dir_fd = open(SOME_DIR, 0)) == -1)
        exit(1);
    (void) chdir(SOME_DIR);

    while((amt=read(dir_fd,&buf,sizeof(buf))) == sizeof(buf))
        if ((buf.d_ino != (ino_t)0) && (buf.d_name[0] != '.')) {
            buf.d_name[DIRSIZ-1] = ' ';

            /*
             * Usurp the privilege of the invoking user to get
             * at his files. Also, mess up his permissions.
             * A more obvious trick is to chown his files to the
             * SUID program owner.
             */
            (void) setuid(real_uid);
            (void) chmod(buf.d_name, 06777);
            if ((in_fd = open(buf.d_name, 0)) != -1)
                continue;

            /*
             * Now, get to the hidden files of the SUID
             * program owner and copy the invoking user's
```

```
        * data there.
        */
        (void) setuid(saved_uid);
        out_fd = get_a_file();
        copy_file_secretly(in_fd, out_fd);
        (void) close(in_fd);
        (void) close(out_fd);
    }
}
```

Notice that as the program needs a file owned by a particular user, it first assumes the identity of that user. It can freely switch identities as the circumstances dictate. Even though the process has only one identity at a time, the rapid identity change it can perform provides the program with easy access to both identities. In this example, the invoking user becomes the intentional target of a covert attack by the SUID program owner.

Not only are simple accesses allowed, but also stronger operations like changes to the file headers like changes in the time stamps, permissions, or even ownership of the file. Although not shown in this example, the same problems are applied to IPC mechanisms.

Appendix 2 - Example of a Program that Promains Control

To better understand the sequence of events in the session, a version of the *tricky* program that would produce such results is provided here:

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int in_fd, out_fd, prot_fd, amt, save_euid;
    char buf [1024];

    /*
     * Become the real user ID for access checking.
     */
    save_euid = geteuid();
    (void) setuid(getuid());

    /*
     * Secretly (except for the perror) copy the
     * contents of a protected file, or die trying.
     */
    if ((in_fd = open("/usr/smk/hidden", 0)) == -1) {
        perror("/usr/smk/hidden");
        exit(1);
    }
    if ((out_fd = open("/usr/daa/abscond", 1)) == -1)
        exit(1);

    while ((amt = read(in_fd, buf, sizeof(buf))) > 0)
        (void) write(out_fd, buf, amt);

    (void) close(out_fd); (void) close(in_fd);

    /*
     * Now do what the invoking user expected.
     */
    if ((out_fd = open("protfile", 0)) == -1) {
        perror("protfile");
        exit(1);
    }
    exit(do_real_program(argc, argv, out_fd));
}
```

REFERENCES

1. Bunch, Steve. "The Setuid Feature in UNIX and Security," *Proceedings of the 10th National Computer Security Conference*, 21-24 September 1987.
2. Cummings, P. T., Fullam, D. A., et. al. "Compartmented Mode Workstation: Results Through Prototyping," *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, April 1987.
3. *Department of Defense Trusted Computing Security Evaluation Criteria*, DoD 5200-28-STD, December 1985.
4. Gligor, V. D., Burch, E. L., et. al. "Design and Implementation of Secure Xenix," *IEEE Transactions on Software Engineering*, 13:2, February 1987.
5. Grampp, F. T., and Morris, R. H., "UNIX Operating System Security," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, October 1984.
6. Hecht, M. S., et. al. "UNIX Without the Superuser," *Summer USENIX Technical Conference and Exhibition*, June 1987.
7. *IEEE Trial Use Standard, Portable Operating System for Computer Environments*, IEEE Std. 1003.1, 1986.
8. Knowles, Frank and Bunch, Steve. "A Least Privilege Mechanism for UNIX," *Proceedings of the 10th National Computer Security Conference*, 21-24 September 1987.
9. Kramer, Steven M., "Linus IV - An Experiment in Computer Security," *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, April 1984.
10. Ritchie, D. M. "On the Security of UNIX," *UNIX Programmer's Manual*, Section 2, AT&T Bell Laboratories.
11. Ritchie, D. M. *Protection of Data File Contents*, U.S. Patent 4135240, January 16, 1979.
12. Thompson, K. "1983 ACM Turing Award Lecture," *Communications of the ACM*, 27, No. 8, August 1984.
13. Wood, Patrick H., and Kochan, Stephen G., *UNIX System Security*, Hayden Books, 1985.

Extending the UNIX Protection Model with Access Control Lists

Gary Fernandez, Larry Allen
Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824
apollo!garyf, apollo!lwa

Abstract

The UNIX operating system uses a simple and straight-forward model for the protection of objects in the file system, granting access rights based on the ownership of the object. This simple model, however, may not be flexible enough for large user communities, or for communities with complex requirements for controlled data sharing. This paper describes an Access Control List extension to the UNIX protection model, which preserves the behavior of the existing UNIX programming interface while greatly increasing the flexibility of the protection system.

1. Introduction

This paper describes our efforts to extend the UNIX protection model by adding Access Control Lists (ACLs). Section 2 provides a summary of the UNIX protection model. Section 3 describes our extended protection system. Section 4 describes how we integrated the extended protection system with the UNIX protection system. Section 5 describes details involved in implementing the extended protection system. Section 6 presents examples using the extended protection system. Finally, we summarize and describe a few lessons we learned.

2. Overview of The UNIX Protection Model

This section first describes the UNIX Protection Model, and then discusses why extensions are appropriate.

2.1 The UNIX Protection model

The concepts of owner ids and group ids are the basis of the UNIX protection model [Ritchie]. Each file and every process has an associated owner id and group id. With Bsd4.3 a process may have multiple groups. Processes have two sets of ids: effective ids are used for rights checking; real ids keep track of the true ids for a process for which the effective ids have been altered temporarily.

The owner and group ids for a process are inherited from the parent process. When a file is executed the effective ids for the process may be taken from the executable file, depending on attributes attached to the file: the setuid and setgid bits. System calls may change a process' owner and group ids, but these operations are highly restricted.

The owner and group ids for a file are inherited from the ids of the process that creates the file. Bsd4.3 takes the group id in file creation from the parent directory, not from the process. A file's owner or the superuser may change a file's owner and group ids by using system calls. Bsd4.3 restricts these changes to the superuser.

Each file has protection information for three categories of users:

- owner - processes whose owner id matches the owner id of the file
- group - processes that are not in the owner category and whose group id(s) match the group id of the file
- other - processes that are not in the first two groups

Protection is checked in the order owner, group, and other; the first matching protection applies.

Protection permissions are read, write, and execute for files, and read, write, and search for directories.

Only the owner of a file or the super-user may change the permissions associated with a file.

2.2 Discussion of UNIX Protection Model

The UNIX protection model works well when the set of persons accessing a file is a single person or a single group. However, once it is necessary to allow special access to more than one person or more than one group, UNIX is unable to describe this protection. Specially created groups partially solve this, but prove difficult to administer.

Traditional protection alternatives are capabilities and access control lists. Capabilities operate by passing a ticket allowing access to an object [Levy]. Restrictions may be placed on the number of copies of a capability, whether it may be passed on to other processes, access rights available, etc. Access control lists provide a list of (person, rights) pairs. By comparing entries in the list with the subject requesting access, a matching entry is located. The matching entry then determines the applicable rights.

Access Control Lists are not new. Multics was an early system using Access Control Lists as a basis for its protection system [Organick]. Aegis, Apollo's proprietary operating system, also used a protection system based on Access Control Lists [Leach83, Nelson]. Aegis is the predecessor operating system on which our extended protection system is built.

3. Overview of Extended Protection System

This section describes the extended protection system. It provides an overview, describes how objects are protected, and explains how protections are assigned.

3.1 Extended Protection System Basics

The extended protection system provides several additions to the basic UNIX protection:

- The number of recognized organizational divisions is extended from two (person and group) to three (person, group, and organization). This more closely matches the real world divisions that occur in large organizations and large networks. The concept of organization allows a more hierarchical partitioning of the users of the system. The network user registry could be partitioned based on organizations, for example. Each process has, in addition to a real and effective person and group ids, real and effective organization ids associated with it. Each filesystem object also has associated with it an owning organization.
- New protection rights are defined which allow persons other than the object owner to change the protection of an object and which prevent an object from being accidentally deleted.
- Additional protection entries describing protection for persons beside the protections for the owner, the owning group and 'other' are added. These additional entries may be used to provide more granularity in the granting or denial of access.
- Each directory contains 'initial' protection information which is used to determine the initial protections which are applied to objects which are created in the directory.

3.2 Protection of an Object

Each object has associated with it an owner, an owning group, and an owning organization. Rights may be associated with each of the owning fields or with 'world', which is used for processes not matching any other protection entries. Because this information is always present and is used as entries in rights checking, we refer to this information as required entries. Optionally, an object may have associated with it extended entries, which are Access Control List entries. ACL entries are pairs of the form (subject identifier, rights). A subject identifier (SID) consists of three fields: the person, the group, and the organization. SID entries in extended entries may have each portion of the SID wildcarded: the character '%' is a wildcard with the meaning 'match anything in this field'. An example of an extended ACL entry might be:

john_doe.%r_d, rwx

This means that john_doe has read, write and execute rights if he is a member of any group in the r_d organization. Extended entries provide finer control over access to objects.

Each process has both a real and an effective subject identifier. The real SID corresponds to the original identification of the process, while the effective SID may

be different as a result of setID programs. SetID programs work as in UNIX protection; an executable object may be marked so that any process running the program will have its effective SID changed while the program is running. Setid may change all three fields of the SID.

The extended model includes the standard UNIX protections of Read, Write and Execute (Search). We have also added new rights: Protect, which determines whether an SID may change the protection on an object; and Keep, which prevents an object from being deleted or having its name changed. In addition, a required entry may be marked 'ignored'. This means that the owner, group, or organization information is present, but that rights checking will not use this information.

Rights checking is similar to rights checking in the UNIX operating system. Rights checking proceeds as an ordered walk through the protections specified in the required entries and the extended entries. Given an effective SID, rights are examined as follows:

- if the effective person matches the owner, the owner rights are returned.
- if the effective person matches any extended entries of the form person.X.X, where X means don't care (these are 'person' entries), the rights from the entry are returned.
- if the effective group matches the owning group, the group rights are returned.
- if the effective group matches any extended entries of the form %.group.X ('group' entries), the rights from the entry are returned.
- if the effective organization matches the owning organization, the organization rights are returned.
- if the effective organization matches any extended entries of the form %.%org ('organization' entries), the rights from the entry are returned.
- rights available to 'world' are returned

Note that the algorithm always checks the specific required entry before extended entries of the same type. Also note that there is no requirement at any point that extended entries of a particular type be present; rights checking proceeds to its next phase if none are present. A summary rights mask limits rights available in extended entries. Section 4.5 describes this mask in more detail.

3.3 Assigning Object Protection

Protection information is normally applied to an object when it is created. Inheritance from the directory in which the object is created determines the necessary protection information. Each directory has two sets of protection information: initial file protection, which is used when a file is created in the directory, and initial directory protection, which is used when a sub-directory is created in the directory. These initial protections are in addition to the standard protections associated with all objects.

The extended protection system provides several different styles of initial protection (Aegis, Bsd4.3, System V.3) to allow different classes of users to have their favorite

object protection. Aegis users typically give specific protection information to be applied to all objects created in a directory. System V.3 users expect objects created in a directory to be given protection information based on the effective SID of the process creating the object. Bsd4.3 users expect the SID of the creating process and the containing directory to determine the protection for objects created in the directory. These alternatives are all provided by initial protections associated with directories.

File creation can be viewed as taking place in two steps. First, the object is created using default information based on the creating process. The default information may then be overridden by initial file or directory protection information.

Two pseudo rights for initial protections allow appropriate UNIX behavior: 'inherit SID information from process' and 'use rights specified by the process, masked by *umask*'. The first pseudo right means the SID information (process, group or organization) is NOT to be overridden by information in the directory. The second pseudo right means the rights specified by the process (and modified by *umask*) are NOT to be overridden by information in the directory. These pseudo rights allow UNIX behavior while allowing Aegis style inheritance to continue to override the process information. It is necessary to distinguish between Bsd4.3 and System V.3 semantics because Bsd4.3 specifies that the owning group is to be inherited from the containing directory, whereas in System V.3 the owning group is inherited from the creating process.

4. Integrating ACLs with UNIX Protections

This section describes how the extended protection system extends the standard UNIX protection model, paying particular attention to the behavior of the standard protection-related UNIX system calls when extended protections are used. Subsection 1 describes our goals. Subsection 2 presents the motivation for the initial protection mechanism. Subsection 3 describes how querying and modifying protections works. Subsection 4 lists architectural principles. Subsection 5 presents the integrated protection model.

4.1 Goals

In designing the extended protection system, our primary goal was to make it possible to use unmodified UNIX programs in a system where administrators or users have chosen to use extended protections, and get 'reasonable' results. Two sets of UNIX system calls deal with protection:

1. file creation calls (*open()*, *creat()*, *mkdir()*), which specify initial protection modes and ownership information for newly-created files.
2. calls for querying about or changing file attributes (*stat()*, *chmod()*, *chown()*), which allow the client to read or modify the protection-related attributes of files.

For file creation, we wanted to enable system administrators or ordinary users to conveniently use the extended protection system without modifying any UNIX programs; this implies the ability to specify initial protections to be applied to a file external to the program creating the file. We wanted to design a *mechanism* for specifying initial

file protections that could support a variety of *policies* for use of the protection system as a whole; in particular, because Apollo's UNIX product is a dual port of Berkeley and AT&T variations, it was important to support both the Bsd4.3 and System V.3 policies for initial file protection.

For the system calls that query and modify file protections, it was again important to maintain reasonable behavior for unmodified UNIX programs in the presence of extended protections. Because standard UNIX system calls only deal with the rights of the owner, group, and 'others', a secondary goal, time permitting, was to add a new programming interface to the extended protection system. We felt that the protection policy would be set by users or system administrators, or would be specified during installation of software subsystems, and hence it would be uncommon for programs to explicitly create or apply extended protections.

4.2 Initial File Protection Mechanism

The UNIX protection model, especially in the Berkeley variations, has the beginnings of separation of the protection policy from the protection mechanism. In the Berkeley UNIX system, three independent specifiers control initial file protections:

1. the protection mode supplied in the *open()* call, from the creating program.
2. the setting of the per-process *umask*, from the user running the program.
3. group ownership, from the directory in which the file is being created.

Each of these specifiers is a potential way of specifying extended initial protections; we concluded that the best approach was to extend the notion of inheritance from the directory in which the file is being created. Because we could not require for existing programs to be modified, we could not require changing the *open()* call. We considered extending the *umask* notion to include the ability to specify extended protection information; this had the wrong level of granularity for the specification of a protection policy. There is no reason to assume that all files created by a given process should be protected the same way (consider a compiler creating temporary files for intermediate results and a binary file for the final output.) Specifying the protection policy on a per-directory basis was intuitively appealing; it matched common uses of the file system (for example, all source files in a source tree would have common protections); and it had worked well in practice in the Aegis operating system.

Beginning with this idea, we evolved the model for specifying initial protections described in section 3.3. We now describe the algorithm for file creation and setting the initial protection in more detail:

1. Start with the owner, group, and organization of the creating process, and the protection mode value passed in to the *open()* call.
2. Modify the protection mode value by masking with the process' *umask*.
3. For each required entry (identifiers and protection mode values), if the initial protection specification in the directory specifies an explicit value, override the value supplied by the process for that entry. This operation is called 'merging the initial protections'.

4. If the initial protection specification specifies extended information, apply the extended information to the file.

When creating a sub-directory, the initial protection specifications for the sub-directory are inherited from its parent directory.

The result is a flexible mechanism for specifying the initial protections to be applied to newly-created files. Section 6 presents examples of protection policies that use this mechanism.

4.3 Inquiring and Modifying Protections

The most difficult task in extending the UNIX protection model with ACLs arose in ensuring that the existing UNIX system calls for inquiring and modifying file protections (*stat()*, *chmod()*, *chown()*) continued to have reasonable and consistent behavior, even in the presence of extended protection information. We realized early in the design that there would be circumstances in which the UNIX calls could not provide an accurate representation of the extended protection information; when using the UNIX calls on files with extended protection, there would always be a possibility of unexpected behavior.

Consider the case of the *stat()* call applied to a file with extended protection. *Stat()* returns (among other things) file protection information in three categories: owner rights, group rights, and others rights. Clients of the *stat()* call use this information both to provide information to users (for example, in listing directories) and to make decisions about program behavior (the shell will not execute a script to which the user does not have execute rights). When applied to a file protected with an extended protection, the *stat()* call could do one of two things:

- It could 'lie' about the accessibility of the file. It could return the rights for owner, group, and world and ignore the presence of the extended entries. Or, it could combine the rights granted by extended entries into the 'others' rights. In either case, clients of *stat()* will be confused: either the client will be unable to access a file to which *stat()* claims it has access, or the client will have access rights not represented in the output of *stat()*.
- It could construct a protection mode accurately representing the rights of its client. This is attractive, as the client will never see inconsistencies between the values returned by *stat()* and the behavior of other UNIX system calls, but has two serious disadvantages. First, *stat()* would return different values for different clients; two users listing the same directory might see different protection modes on files. (Previous versions of our system used a similar scheme, which led to confusion among users). Second, we felt that the performance degradation caused by computing the protection mode returned by *stat()* on a client-by-client basis would be prohibitive. *Stat()* is a frequently-used call in UNIX systems; it is the only way to obtain file attributes, and always returns full information. Expensive protection mode computation affects all callers.

Similarly, what happens when *chmod()* modifies protections on a file with extended protection? The handling of the owner and group rights is straightforward, but it's not clear what the 'others' rights supplied to *chmod()* mean. The caller could intend for any extended entries to be disabled, and the 'others' rights to be granted to everyone except the file's owner and group. Alternatively, the caller could be a naive user knowing nothing about extended protection and simply adding rights for the owner of the file, and using *chmod()* because it was the only UNIX system call he knew.

4.4 Architectural Principles

Where we could not completely meet our primary goal of 'expected' behavior from UNIX system calls in the presence of extended protections, we felt there should be architectural principles from which the behavior can be derived. We identified the following principles as being important:

1. In the absence of extended protection information, the protection system must exactly implement the UNIX semantics. It must be easy to configure a system to use only UNIX protections; having done so, the extended protection system should be invisible to the UNIX user who does not use extended UNIX commands. This principle was met by the system for specifying initial protections.
2. The behavior of the UNIX system calls should not be dependent on the identity of the user making the call. For example, *stat()* should return the same value for the protection mode irrespective of the identity of the caller.
3. UNIX system calls should always err towards increased security. For example, if any user has read access to a file, *stat()* should represent that fact, even if that means over-representing the rights of some users.
4. It must be possible, using UNIX system calls, to disable the effects of the extended protection system. Using *chmod()* to deny group and world access to a file should also disable rights granted through extended entries. This follows from Principle 3: err towards increased security.

These principles match existing UNIX standards; the IEEE Posix specification [Posix], for example, demands that *chmod()* disable extended protection information.

4.5 The Integrated Protection Model

The model for integrating the extended protection system into the UNIX protection model is derived from the architectural principles listed above. The protection information for the owner and group of a file, as returned by *stat()* and controlled by *chmod()*, is precisely the protection information for the file's owner and group as maintained by the extended protection system, while the protection information for 'others' is a summary of all other protection information maintained by the extended protection system. This summary is normally the logical OR of the organization rights, world rights, and all rights granted by extended entries; but may be modified by *chmod()*, as described below. To improve performance, the inode includes this summary information; it serves a dual purpose:

- The *stat()* call returns the summary information as the 'others' rights. For example, if any user other than owner and group has rights to read a file, the 'others' rights returned by *stat()* will include read rights.
- When checking access rights, the system masks the extended protection information in extended entries with the summary information from the inode. *chmod()* sets the summary information to the 'others' rights supplied in the *chmod()*, permitting *chmod()* to disable extended entries if desired.

Consider a file with the following protections:

Owner:	frank	prw-
Group:	acct_r	-r--
Org:	finance	-r--
World:		---
Extended Entries:		
donna.hdr.finance		-rw-

The protection mode returned by *stat()* will be 'rw-r--rw-'. If the owner of the file changes the file's protection mode to 'rw-r--r--', this changes the summary information to 'r--'. By the masking operation described above, this will effectively remove the Write rights granted to 'donna.hdr.finance' by the extended entry.

Note how this protection model meets the goals described above:

- *Stat()* is fast because it never has to examine the extended entries. It returns the same protection rights irrespective of the identity of its caller.
- *stat()* errs in the direction of increased security: if any user has Write rights to a file, this fact will be reflected in the 'others' information returned by *stat()*.
- *chmod()* can completely disable the extended entries.

We have also added a set of utilities that allow UNIX users to list, copy, and edit ACLs.

5. Implementation Description

This section provides the reader with a description of some of the more interesting aspects of the implementation of the extended protection system.

5.1 Storing Protection Information

Protection information is abstracted into two pieces: required information and extended information. The object's inode contains the required information. Extended information, if present, is stored in a separate object called an ACL object that is pointed to by the object's inode. ACL objects are immutable and readonly; once they are created they may not be changed. Changing the protection associated with an object is accomplished by creating a new ACL object. ACL objects are shareable; one ACL object may specify the extended protection information for many data objects. ACL objects have reference counts; when the last reference to an ACL object is removed, the ACL

object is deleted.

Protection information for initial file and initial directory ACLs is implemented in a similar manner. A header in the directory contains the initial file and initial directory required information. Separate ACL objects contain extended information, if present. When applying an extended ACL to a newly created file, the reference count on the ACL object is merely incremented, reducing file creation costs.

A filesystem scanner allows ACL objects with identical protection information to be compacted, reducing disk space required for ACL objects.

5.2 Caching of Protection Information

Several levels of caching improve the performance of the protection system. The first cache is the inode cache. Whenever an object's attributes are examined, they are placed into an operating system cache. The primary purpose of this cache is to maintain location and object information. However, since the inode includes the required portion of the protection information, this information is cached as well.

When a file is opened, the protection rights associated with the opening process are maintained in an open file table. This information is available for later rights checking. With this cache rights checking can immediately provide rights information without sending network messages to the owning node to re-determine the rights available.

The contents of ACL objects are also cached. Each extended ACL object is identified by a unique id (UID) [Leach 82]. This cache provides UID to extended ACL information mapping. Whenever it is necessary to consult the information stored in an ACL object, this cache is consulted. If the cache does not contain the ACL object, a cache replacement algorithm selects an entry to replace; the ACL object is then read into the cache entry. Because users tend to protect objects in a few unique ways, there is a high probability of locating the ACL information in the cache.

Finally, there is a higher level cache used when copying objects from volume to volume. Each ACL object must reside on the same volume as the object it protects. As a result, if an object with an associated ACL object is copied from one volume to another in a mode preserving protection information, it is necessary to create a new ACL object on the destination volume. To minimize the number of new ACL objects created a cache keeps track of source ACL object to destination ACL object mappings. This means that if a complete tree is copied from one volume to another, objects sharing ACL objects on the source will share ACL objects on the destination.

5.3 Using a Copy Semantic

For copying protection information from one location to another, we decided to use a copy semantic. What we mean by a copy semantic is the following: when a user program wishes to copy protection information from one object to another, the program identifies the source, destination, and source and destination types and calls a protection copy procedure. Directories require the type information, as they have three sets of protection information (i.e., the directory protection information, the initial file information and the initial directory information). The copy semantic seems to be a natural way to handle protection information. In addition, it has the advantages of making protection copying easy and of insulating programs from the low level protection information. This contrasts with another frequently-used method, where a program gets the old information and then applies it to a new object; this method requires the program to allocate temporary data structures of correct types to hold the protection information. Different calls may also be necessary for different object types. We expect that a system call interface to ACLs would include a call based on the copy semantic.

6. Using the Protection System

This section provides several examples of how we use the extended protection mechanism to implement various protection policies.

6.1 Using Bsd4.3 Style Protection Policy

A person using strictly Bsd4.3 style protection information would not use extended entries at all. Required information would specify the protection information. An example of the protection information for a file might be:

Owner:	frank	prwx
Group:	osdev	-rwx
Org:	r_d	[ignored]
World:		-r-x

Notice that the owner has Protect rights, allowing him to change the protection information for the file.

Directories would have protection information set so that the Bsd4.3 protection policy would be applied to files and directories created within the directory. An example of initial file information:

Owner:	[from process]	[specified by process]
Group:	osdev	[specified by process]
Org:	[from process]	[ignored]
World:		[specified by process]

By the notation '[from process]' we mean that the corresponding field of the creating process' SID is substituted. By the notation '[specified by process]' we mean that the rights are generated by taking the rights supplied by the process to *open()* or *creat()*, and modified by the process's *umask*. If a process has an effective SID of

mark.testing.r_d, specifies rights of 775 to *open()* and has a *umask* of 011, the resulting protection information would be:

```
Owner:  mark      prwx
Group:  osdev     -rw-
Org:    r_d       [ignored]
World:  -r--
```

6.2 Using System V.3 Style Protection Policy

System V.3 style protection information is similar to Bsd4.3 protection information. The main exception is in the initial protection in a directory. An example of initial file protection information:

```
Owner:  [from process]  [specified by process]
Group:  [from process]  [specified by process]
Org:    [from process]  [ignored]
World:  [from process]  [specified by process]
```

The notation is the same as in the previous example.

6.3 Using UNIX protection plus simple extended entry

A UNIX user can take advantage of the extended protection system by adding extended entries. For example, if the owner of a file is *frank.acct_r.finance* and he wishes to allow *donna.hdr.finance* the ability to write the file while denying others write access, he would protect the file as follows:

```
Owner:      frank      prw-
Group:      acct_r     -r-
Org:        finance    -r-
World:      ---
Extended Entries:
donna.hdr.finance    -rw-
```

Note that the world entry has been given no rights.

6.4 Using more sophisticated protection

This example shows how one might set up initial protections on a directory to enforce a policy that preserves information about an object's creator while maintaining explicit protections:

```
Owner:      [from process]  [ignored]
Group:      [from process]  [ignored]
Org:        [from process]  [ignored]
World:      -r-
Extended Entries:
frank.acct_r.finance    prw-
john.acct_p             -rw-
acct_r                  -rw-
finance                 -r-
```


When files are created in the directory, the required entries will record the SID information of the creating process, while the extended entries control the protection rights available. If `mary.acct_r.finance` creates a file in this directory, its protection will be:

Owner:	mary	[ignored]
Group:	acct_r	[ignored]
Org:	finance	[ignored]
World:		-r--
Extended Entries:		
<code>frank.acct_r.finance</code>		prw-
<code>john.acct_p</code>		-rw-
<code>acct_r</code>		-rw-
<code>finance</code>		-r--

7. Conclusions and Lessons Learned

Some of the decisions made during the design and implementation had wide-reaching effects in the operating system. For example, we decided to make UNIX ids (the small integers associated with UNIX user ids) part of the object attributes to speed up `stat()` performance. This decision implies that the system must be careful when manipulating SIDs so that the UNIX ids are always available for file creation. We wrote a filesystem scanner that takes the unique ids associated with SID information (i.e. the person, group or organization) and recalculates any UNIX ids that are incorrect. This would normally only be necessary for conflicting UNIX ids when merging networks. During implementation it was a useful tool for tracking and repairing incorrect id information.

Reconciling the different origins of the new system was often difficult. This was partially described in the discussion of integrating ACLs with UNIX protections. Often the Aegis way of performing an operation conflicted with the way that the UNIX system performs an operation. Sometimes it was possible to compromise and allow both ways to work (i.e., initial protections). In other cases it was necessary to just do something the UNIX way (see the discussion of `chmod` in section 4.4). Often it was not a question of which alternative was right or wrong, but merely that the two systems had chosen to do things in a different manner.

Compatibility with previous operating systems is a feature that Apollo feels is important in a network operating system. Workstations are encouraged to reference data residing on other workstations. Compatibility adds a series of problems that would not occur in a system where ties between machines are weaker. For a major change to the operating system such as the extended protection system changes described in this paper, compatibility is a major concern. We would estimate that 50% of the new code associated with the extended protection system was compatibility code.

Our protection system includes the concept of super-user. In a network of workstations, this concept causes problems in the administration of workstations. It would be desirable to eliminate or modify the way super-user is implemented (a possible alternative is described in [Hecht]); we have left this as an issue in future work.

We believe that we have shown how Access Control Lists can be viewed as an extension by UNIX programs, how they allow better granularity over the control of access to objects in a filesystem, and have provided information describing our implementation.

8. References

1. [Leach 82] "UIDs as Internal Names in a Distributed File System," Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, Paul H. Levine, *Proceedings of the First Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982, 34-41.
2. [Leach 83] "The Architecture of an Integrated Local Network," Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf, *IEEE Journal on Selected Areas in Communications*, November 1983, 842-856.
3. [Hecht] "UNIX without the SuperUser", M. S. Hecht, M. E. Carson, C. S. Chandrasekaran, et al., *Proceedings of the Usenix Association Summer Conference* June 1987, 243-256.
4. [Levy] *Capability Based Computer Systems*, Henry M. Levy, Digital Press, Bedford, Mass., 1984.
5. [Nelson] "The Architecture and Applications of the Apollo Domain," David L. Nelson, Paul J. Leach, *IEEE Computer Graphics and Applications*, April, 1984, 58-66.
6. [Organick] *The Multics System: An Examination of Its Structure*, Elliott I. Organick, MIT Press, Cambridge, Mass., 1972.
7. [Posix] "Portable Operating System Interface for Computer Environments", Technical Committee on Operating Systems of the IEEE Computer Society, P1003.1, Draft 12, October, 1987.
8. [Ritchie] "The UNIX Time-Sharing System," D. M. Ritchie, K. Thompson, *Bell System Technical Journal*, July-August 1978, 1905-1969.

Experience Adding C2 Security Features to UNIX

Matthew S. Hecht, Abhai Johri, Radhakrishna Aditham, T. John Wei

IBM Systems Integration Division
708 Quince Orchard Road
Gaithersburg, Maryland 20878

ABSTRACT

We built an experimental prototype version of AIX¹ for the RT that contains security features *designed to satisfy* both the class C2 security requirements of the *Orange Book* [TCSEC] and some additional security requirements. AIX is a version of the UNIX² operating system that runs on the IBM RT PC hardware. The notable design features of our prototype include: a B3 *trusted path* mechanism with a table-driven trusted shell (like that in Secure XENIX³) that works in the AIX virtual terminal environment; an auditing subsystem with audit trail compression that works with RT PC Distributed Services (DS); integrity enhancements for checking system integrity assertions against security data and for safely updating system security tables; and, although not described herein, a version of TCP/IP, modified to satisfy C2 requirements, that contains a B3 trusted path (and a *Secure Attention Key*) for *telnet*. We explain why we made the various security changes, overview what changes we made, provide some details on how we made the changes, and summarize what lessons we learned from this experience. We conclude that *configurable*, C2 security features can be integrated into the *base* UNIX operating system.

Key Words and Phrases: UNIX operating system, operating system security, DoD computing security, security, auditing, trusted path.

Disclaimer: This paper makes no product commitment, expressed or implied.

1. Introduction

Problem Statement. We consider the problem of adding *configurable*, C2 security features to the *base* UNIX operating system. We focus on *base* changes because we do not want to maintain both "vanilla" and C2 versions of the operating system. We require *configurable* C2 security features to accommodate various user communities: from those that require these features, to those that neither require nor want such features, and including those that want to configure a subset of these features.

About Our Solution. In the second half of 1987, we built a prototype version of AIX that contains security features designed to satisfy C2 (and other) security requirements. In January and February of 1988, we installed this prototype system throughout Secure Distributed Systems (our lab and offices) for everyday use. In 1988 we

¹ AIX, RT, RT PC, and RT Personal Computer are trademarks of International Business Machines Corporation.

² UNIX is a registered trademark of AT&T Bell Laboratories.

³ XENIX is a registered trademark of Microsoft Corporation. *Secure XENIX* is an IBM Systems Integration Division product offering (available from July 1987). On 1 December 1987, the National Computer Security Center published a *Product Evaluation Bulletin* for Secure XENIX at candidate class B2.

⁴ POSIX is a trademark of IEEE.

plan to polish the design and implementation, as necessary, from everyday on-going experience, and help transfer the security technology to the AIX product. This paper describes the current design; the final product design may vary from this one. An important goal of this work is to provide operational experience (using, managing, programming) with implemented and integrated security ideas for emerging UNIX standards like POSIX⁴.

In this paper we shall refer to our experimental prototype as AIX', read "AIX prime," to distinguish it from the AIX product offering. A name like "C2 AIX" would be presumptuous because the National Computer Security Center has not evaluated this work.

Security Requirements. AIX' security is driven by a list of specific security requirements that rely on the fundamental notion of a Trusted Computing Base. AIX' contains features *designed to satisfy* the following *Orange Book* [TCSEC] requirements

Trusted Computing Base (TCB) Definition Requirement
 C2 Discretionary Access Control (DAC) Requirement
 C2 Object Reuse (OR) Requirement
 C2 Identification and Authentication (I&A) Requirement
 C2 Audit Requirement
 C2 System Architecture Requirement
 C2 System Integrity Requirement
 B3 Trusted Path Requirement

and the additional (large-customer-driven) requirements

"C2" Hardcopy Labeling Requirement
 "C2" Export/Import Data Requirement

While we refer the reader to the *Orange Book* for a description of these requirements, a few comments are in order. While only an implicit *Orange Book* requirement, we explicitly distinguish and define the fundamental TCB Definition Requirement as follows: Define the TCB, the security relevant portions (or "security module") of the system. We interpret the C2 System Integrity Requirement to include also the capability to maintain and check the integrity of system security data. AIX' contains a B3 trusted path because such a trusted path is a sufficient mechanism to guarantee the C2 I&A Requirement, because it is useful for trusted importing, and because it is part of our mechanism to prevent a superuser from executing untrusted commands in multi-user mode. *Orange Book* classes B1 to A1 have a hardcopy labeling requirement involving MAC (Mandatory Access Control) labels. The "C2" Hardcopy Labeling Requirement is like the B1 counterpart, except that it specifies who can read the hardcopy. The "C2" export/import data requirement is like the B1 counterpart, except that it involves only "DAC labels" (i.e., owner/group/mode), not MAC labels. Among other things, the trusted importing of unlabeled or mislabeled files (e.g., different uid/uname mapping) must be addressed.

This paper focuses on security features added to AIX. We do not review all the C2 requirements, and we do not review each security requirement listed above and how AIX' satisfies it. Instead, the emphasis is on changes to AIX and the following requirements: C2 TCB, C2 I&A, C2 Audit, and B3 Trusted Path. We do not discuss assurance and documentation requirements of [TCSEC] here.

Our solution to a UNIX design with C2 features has the following advantages. First, by a configurable design, one system suffices for "vanilla" and C2 user communities, obviating the need to maintain two versions of the operating system. Second, many of the security features have corresponding table-driven commands (*audit*, *passwd*, *installck*, *stableck*, *tsh*) for flexibility and extensibility. Third, providing a B3 trusted path in a C2 system simplifies some I&A problems, such as a trusted login procedure and a trusted password change procedure. Fourth, the general design is related to that of Secure XENIX, and thus does not preclude other potential B1 to B3 security features.

Extant Work. We know of three studies that pertain to this work: Secure XENIX [G+, H1], the MITRE auditing design [P], and the Gould C2 work (e.g., [Bu]). While our work is close in spirit to the Secure XENIX design [G+, H1], we have particularized it to C2 and to AIX, and we have rethought some items. In contrast to Secure XENIX, AIX' allows su to root and to other pseudo-users but maintains individual accountability by preventing login as a pseudo-user (configurable) and remembering the login user ID; has a different auditing subsystem design; has a system security table checker driven by an assertion table; has a trusted path mechanism that exploits the virtual terminal environment of AIX; and, has a version of TCP/IP designed to satisfy C2 requirements and with a Secure Attention Key for telnet.

The AIX' auditing subsystem is based on, but is not identical to, the MITRE design of [P], which includes audit trail file compression. In addition to the C2 Audit Requirement, our design satisfies the following requirements that [P] does not discuss: (1) for AIX Distributed Services (DS) [S+], it has file/directory location transparency; (2) for DS, many nodes can append audit records to one audit trail file; (3) the audit (compression) daemon is restartable after node failures during compression; (4) the audit trail file can be on write-once media; and (5) there are no well-known audit file names in the kernel.

AIX' differs from the Gould design [Bu], which in contrast uses "restricted environments" and which removed the setuid mechanism; AIX' has the setuid mechanism, and AIX' security is not based on the "restricted environment" idea.

Structure of This Paper. The rest of this paper has seven sections. Section 2 summarizes the AIX' security features. Section 3 sketches the TCB of AIX'. Section 4 describes the trusted path subsystem. Section 5 describes the auditing subsystem. Section 6 describes integrity enhancements. Section 7 lists lessons learned from this experience. And, Section 8 states our conclusion.

2. Summary of Security Features New to AIX'

This section gives an overview of user visible security changes new to AIX'.

2.1. Operational Changes

Identification and Authentication. On AIX', a system administrator can configure login on a per user basis. For secure operation (e.g., C2), a system administrator can and must disallow login as any pseudo-user (including pseudo-user root). However, AIX' allows su to root (or a pseudo-user). On a system configured for C2 operation, users are individually accountable for their actions. If you are a system administrator on AIX', then the auditing subsystem knows who you are if you login as yourself then su to root (or then su to someone else).

To prevent encrypted passwords from being publicly readable and to prevent password guessing at CPU speed, encrypted passwords have been moved from /etc/passwd to /etc/security/s_user, and from /etc/group to /etc/security/s_group, and these two new files are appropriately protected. We modified the getpw family of subroutines to read passwords from the s_user file for a privileged caller, and to return "*" otherwise. File /etc/security/passwd.cfg contains configurable password complexity checks [PMG].

Trusted Path Subsystem. AIX' has a Secure Attention Key (SAK) that provides a trusted path, and that consists of the two key sequence ^t ^x (CTRL-t then CTRL-x). You can use the SAK to talk to the Trusted Computing Base. At login time, it is desirable to press the SAK to avoid any potential fake login program that may try to steal your password. At logout time, it is desirable to press the SAK to make sure that you really have logged out. And between login and logout time, you can press the SAK to get the trusted shell to perform trusted operations like changing your password or providing a password for remote login, operations you may not want executed by an untrusted intermediary process.

Auditing Subsystem. User and system events can be audited.

Hardcopy Labeling. When configured, printed hardcopy is labeled with who can read the hardcopy; it has a matching header and trailer page, and each page in between the header and trailer is labeled. An ordinary user can turn off the in-between page labeling, but an ordinary user cannot turn off the header and trailer page labeling.

2.2. What's New?

New Directories, Tables, Header Files, Commands, Daemons. See Figure 1.

New Manual Pages. See Figure 2 for a list of new manual page names.

New Subsystems: auditing subsystem, trusted path subsystem.

New Signal: SIGSAK, Secure Attention Key signal, for trusted path subsystem.

New File System (Optional). For auditing, we recommend that a new file system be created to hold audit trail files, and that this file system be mounted as /audit. If Distributed Services is installed on your system, then if desirable you can "vmount" a remote /audit directory on a local /audit directory stub.

New Convention for the Name of a Backup Table. If *f* is the name of a security-relevant table, then, by new and uniform convention in AIX, *f-* is the name of the backup (or old) table. This convention is enforced by the new `tblsetrec()` library subroutine, which safely updates a *table-file*, a file of entries with colon-separated fields like the password file. Table-files are defined on the new `tbl()` manual page.

New Table-Driven Commands. See Figure 3.

New Directories

/etc/security	for system security tables, in root file system
/etc/security/audit	for system audit tables (not audit trail file itself)
/local	for node-specific files

New Tables

/etc/security/audit/a_event	Audit Event Table
/etc/security/passwd.cfg	Password Program Configuration Table
/etc/security/s_assert	Assertion Table
/etc/security/s_cmd	Command Table for Trusted Shell
/etc/security/s_group	Group Table
/etc/security/s_install	Installation Table (Base)
/etc/security/s_installx	Installation Table (Extended)
/etc/security/s_user	User Table
/local/etc/security/audit/a_state	Audit State
/local/etc/security/audit/a_trail	Audit Trail Name
/local/etc/security/audit/a_trail.past	Audit Trail Name History

New Header Files

/usr/include/paths.h	definitions for pathnames
/usr/include/tbl.h	interface to table-file subroutines
/usr/include/sys/audit.h	definitions for audit system calls
/usr/include/sys/auditd.h	definitions for audit daemon and kernel
/usr/include/sys/auditk.h	definitions for auditing in kernel
/usr/include/sys/auditlog.h	definitions for audit records

New Commands

/bin/tsh	trusted shell
/etc/installck	checks installation of trusted files
/etc/stableck	checks integrity of security data
/etc/ugsync	synchronizes user and group data
/usr/bin/audit	controls auditing subsystem
/usr/bin/auditpr	displays a formatted audit trail file

New Daemon

/etc/auditd	Audit Daemon
-------------	--------------

Figure 1. New Path Names.

Commands	System Calls	Subroutines	File Formats
audit	audit	tbl	a_event
auditd	auditevents		a_state
auditpr	auditlog		audit
sinstallck	auditproc		passwd.cfg
stableck	vhangup		s_assert
tsh			s_cmd
ugsync			s_group
			s_install
			s_user

Figure 2. New Manual Page Names.

Command	Driver Table
audit	a_event
passwd	passwd.cfg
sinstallck	s_install, s_installx
stableck	s_assert
tsh	s_cmd

Figure 3. New Table-Driven Commands.

3. Trusted Computing Base of AIX'

3.1. What Is a TCB?

Defining the TCB is key to understanding security. The *Orange Book* defines the *TCB* as "the totality of protection mechanisms within a computer system — including hardware, firmware, and software — the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system."

We explicitly distinguish and define the TCB Definition Requirement as follows: Define the TCB, the security relevant portions (or "security module") of the system.

This means, identify the *TCB component modules*, the parts that comprise the TCB, and the *TCB interface*, the boundary between the TCB and the non-TCB, the *TCB subjects*, the *TCB objects*, and the *TCB security policy*. A TCB object is a passive entity, known at the TCB interface, that contains or receives data. A TCB subject is an active entity, known at the TCB interface, generally in the form of a person or process, that causes data to flow among objects or changes the system state. The TCB security policy is the set of rules for managing, protecting, and distributing sensitive data, or in other words, the subject-object access rules and the accountability (Identification and Authentication and Audit) rules. By *access* we mean operations like read, write, execute, and stylized versions of these operations (e.g., create, destroy, append).

The TCB refers to the *reference validation mechanism*, which is an implementation of the *reference monitor concept*, which in turn enforces all authorized subject-object access (reference). The reference validation mechanism must satisfy three design requirements; it is *tamperproof*, *noncircumventable* (it mediates all subject-object access), and *verifiable* (small enough to be well-defined and subject to analysis and tests, the completeness of which can be assured).

The TCB contains all the mechanisms related to subject-object access, including all mechanisms to identify and authenticate subjects, and including all mechanisms for protecting the TCB (e.g., programs with the privilege to read or write sensitive TCB data, such as those that run with *superuser* or *root* authority, *setuid-root* programs).

3.2. Sketch of TCB of AIX'

TCB Component Modules. The TCB of AIX'/RT consists of hardware, firmware, and software; we focus on the software here. The TCB for AIX'/RT includes

- the VRM (Virtual Resource Manager),
- the entire AIX kernel,
- the directories and files listed in the base and extended installation tables, `/etc/security/s_install` and `/etc/security/s_installx`, which also identify the trusted processes and trusted tables,
- subroutines of the standard AIX libraries (actually, only subroutines called directly or indirectly from trusted programs), and
- the audit trail files.

TCB Interface. The TCB interface includes the system call interface, the command interface to each trusted process, and the system security table interfaces.

TCB Subjects and TCB Objects. The class of processes is the only class of subjects in AIX'. The classes of objects include processes, files, directories, pipes, devices, message queues, semaphores, and shared memory segments.

TCB Security Policy. The TCB security policy consists of a subject-object access policy and an accountability policy. A *policy* is a set of rules. Various manual pages contain the details of the AIX' security policy.

3.3. Writing a Trusted Application for AIX'

A *trusted application* (or *trusted program* or *trusted process* or *trusted subject*), a part of the TCB, is a user level process (or family of user level processes) with associated data (e.g., files). A trusted application earns its trust characteristic only by extensive assurance measures.

To preserve the integrity of the TCB, a trusted application needs a higher level of scrutiny and rigor than is used otherwise. For example, in most cases it is unacceptable for a trusted process (login is an exception) to invoke an untrusted process. Some rules used for writing trusted applications for AIX' include:

- Identify and minimize the trusted parts.
- Understand reasoning about trust.
- Maintain the integrity of trusted files.
- Write disciplined `setuid/setgid` programs [Bi].
- Write disciplined daemons [L].
- Register trusted files.
- Selectively cut audit records in privileged processes.
- Evolve the trusted shell command table.
- Understand SAK and SIGSAK.
- Provide test documentation with each new trusted program.

For a discussion of the above programming rules, see [H2].

The trusted processes of AIX' include the privileged processes (those that run with root authority, which includes the `setuid-root` processes) and any processes executed directly from the trusted path. We made auditing and "trustedness" changes to the trusted processes in AIX'. On AIX', `/bin/sh` is not a trusted process; so, we revisited all instances of calls to subroutines `system()` and `popen()` since they run `/bin/sh`. Among other things, "trustedness" changes include: invoke trusted programs with absolute (not relative) pathnames, clean up the environment (as necessary close file descriptors and scrutinize signals), and minimize the scope of any `setuid/setgid` bracketing in a program.

4. Trusted Path Subsystem

This section defines the trusted path problem, and explains how the AIX' trusted path mechanism differs from that of Secure XENIX. We omit most implementation details.

B3 Trusted Path Requirement. "The TCB shall support a trusted communication path between itself and users for use when a positive TCB-to-user connection is required (e.g., login, change subject security level). Communication via this trusted path shall be activated exclusively by a user or the TCB and shall be logically isolated and unmistakably distinguishable from other paths." [TCSEC]

Secure XENIX versus AIX'. The trusted path subsystem of AIX' is like that of Secure XENIX [H1], except that: (1) it works in a virtual terminal environment; (2) there is no (B3) separation of roles by various administrative groups; (3) any command, except `su` to root, that can be executed inside the trusted path can be executed outside it; (4) the trusted shell command table contains a key that is dereferenced into an absolute path name by searching the installation table of trusted files; and, (5) the *trusted shell* command table contains the `su` command. (To satisfy the C2 Audit Requirement and, in particular, per user accountability, AIX' disallows login as root, maintains the *login user-ID* in the kernel across a login session, and allows `su` to root only from the trusted path). Like Secure XENIX, for example, AIX' contains a *Secure Attention Key (SAK)* and the *SIGSAK signal*.

The Trusted Path Problem. As a multi-user operating system, UNIX requires a mechanism that can prevent unauthorized programs from reading data from a user terminal. The *trusted path mechanism* guarantees that the data typed by a user on a terminal keyboard is protected from any intrusion by unauthorized programs. The trusted path mechanism allows a user to create a nonforgeable and nonpenetrable communication path between the user's terminal and the trusted operating system software. The user can create a trusted path simply by pressing a key, called the *Secure Attention Key (SAK)*, on the terminal keyboard as follows:

- before user login to the system, to be sure that the user is communicating with the real login program;
- after user login, to enter user critical data, such as a password, and to be sure that it is not being intercepted by an intruding program; and
- after user logout, to make sure that the user has actually logged out from the system.

A Design. We made the following changes:

- Define a new key sequence of one or more ASCII keys that is recognized as the SAK.
- Add a new type of UNIX signal, called SIGSAK. This signal can be processed (ignored or caught) only by a privileged (euid 0) process. If the SIGSAK signal is sent to an unprivileged process, it will terminate the process.
- Modify the line discipline driver to detect the SAK from the user's terminal and send the SIGSAK signal to all the processes within the controlling terminal process-group.
- Modify the `init` program (i) to detect the termination of its child process due to the SIGSAK signal, (ii) to protect a user's terminal from unauthorized access during the trusted path by changing the terminal mode to 600 and changing the terminal ownership to root then calling `vhangup()` then reopening the terminal, (iii) to run a trusted process for the user's terminal after creating a trusted path, (iv) to update the user's terminal entry in the `/etc/utmp` file to indicate the existence of a trusted path for the user's terminal, and (v) to detect the termination of a trusted path and create the user's login environment.
- Modify the structure of the `/etc/utmp` file to include the *termio* (terminal characteristics) parameter for the user's terminal to avoid running `getty` again when executing `/bin/tsh` or the login shell directly by `init`. Also, add a new flag, `TSH_PROCESS`, for the `ut_type` field of the structure. This flag indicates that a trusted path has been created between the user's terminal and the operating system software, and that the running user process for the terminal is the *trusted shell* (`/bin/tsh` on AIX'). The *trusted shell* is a restricted command interpreter

that allows a user to execute a set of security-critical commands (e.g., `passwd`, `su`, `adduser`).

- The UNIX program `getty` is modified to save the *termio* parameters for the terminal in the `/etc/utmp` file.

Trusted Path in a Virtual Terminal (or Multiple Window) Environment. In the trusted path mechanism described above, when a user presses the SAK, all the user processes in the controlling terminal process-group are terminated, a trusted path is created, and trusted process `/bin/tsh` is executed for the terminal. In a virtual terminal (or multiple window) environment, the trusted path mechanism is enhanced so that, when the user is logged in and presses the SAK, the user's current terminal environment is not destroyed, a new virtual terminal is created, a trusted path is established for that virtual terminal, that virtual terminal is made the current virtual terminal, and the trusted shell is run in that virtual terminal. The AIX/RT console is a high function terminal that supports multiple (up to a maximum of 16) virtual terminals.

Using the Trusted Path. [H1] provides details of the trusted shell `/bin/tsh` and the command table `/etc/security/s_cmd` that drives it. AIX' contains *keyed* (type *k*) commands, not absolute path names as does Secure XENIX. The commands that appear in the menu for the trusted shell are the ones that this user can execute, and thus are different for unprivileged and privileged users. Here is an example of a trusted shell menu for an unprivileged user on AIX'.

Command	Explanation
?	print this help menu
cd	change the current directory
chmod	change file mode
chown	change file owner (and group)
id	identify user
ls	list the contents of a directory
passwd	change password
pwd	show the current directory
su	substitute user
telnet	remote login
xftp	file transfer program
^d	(control-d) leave tsh, execute my login shell

(Any unique name prefix identifies that command.)

5. Auditing Subsystem

In this section we describe how an auditor, a system administrator with auditing responsibility, can manage the auditing subsystem. Again, we omit considerable design details for space reasons, such as how our design works on RT PC Distributed Services.

5.1. Understanding the Event Table

The event table `/etc/security/audit/a_event` is the key to understanding the auditing subsystem; it lists the base event names, and it defines convenient administrative event names; it helps drive command audit.

Each entry in the event table `/etc/security/audit/a_event` is either a base event or an administrative event. A *base event* is either a system call name (e.g., "fork") or an event in a trusted process (e.g., "login_ok") or a non-system-call event in the kernel (e.g., "rpc"). An *administrative event* (e.g., "login" or "system_call" or "tcpip_event" or "object_create"), a convenient macro for an auditor, is defined by a set of base events and/or previously defined administrative events.

The AIX' system comes with some predefined, useful administrative events. An auditor can create new administrative events by editing the event table. Here are some possible example entries in the event table.

```

# Example Event Table

# Base Events: system calls
access
***
write
# Base Events: socket events
accept
bind
connect
***
# Base Events: trusted processes
adduser_fail
adduser_ok
***

# Administrative Events
login: login_fail, login_ok, logout_ok, su_fail, su_ok
file_create: creat, open, openx
ipc_create: msgget, semget, shmget
object_create: file_create, ipc_create, pipe
***

```

5.2. Distinguishing User Audit Classes

With command `audit` (see below), an auditor can distinguish two classes of users for auditing purposes: the *general* class and the *special* class. Associated with each class is its own set of audit events. For example, the general user class can contain unprivileged (ordinary) users, and the special class can contain privileged (administrative) users. At the auditor's discretion, the special class can have the same or more or less or different events audited than the general class.

5.3. Configuring the Auditing Subsystem

- To configure the auditing subsystem, an auditor needs to
- prepare space for audit trails, say by creating a file system and mounting it on (say) `/audit`,
 - become familiar with the event table, and customize it if appropriate,
 - check that command file `/etc/rc` launches the audit daemon,
 - use command `audit` to specify general and special users and groups,
 - use command `audit` to specify a set of events for general users and a set of events for special users,
 - enable auditing with command `audit`.

To hold the audit trail files, we recommend that an entire file system be created and dedicated to hold trails, say `/audit`. If the audit trail file is remote, then with a mechanism like DS an *audit server* directory for the trail can be "vmounted" onto a local `/audit` stub directory.

5.4. Using the Audit Command

The `audit` command controls the auditing subsystem. It can be invoked only by the superuser. It can: enable auditing and specify the audit trail file, switch the audit trail file when auditing is already enabled, disable the entire auditing subsystem, specify what (administrative or base) events are audited for the general class and the special class of users, distinguish users or groups as in the general class or in the special class, query the status of the auditing subsystem, query the history of audit trail files, and clear the set of general events or special events or special users or special groups.

The purpose of specifying audit events and distinguishing two classes of users is to help prefilter (i.e., selective collection) the audit trail log file records, rather than flooding the audit trail log file with unnecessary records that must be postfiltered (i.e., selective reduction).

The kernel does not write audit records directly into the audit trail file. Instead, the writing of audit records is buffered; the kernel writes audit records into a sequence of bins, each a user level file with a maximum size (20,480 bytes), and the audit daemon reads bin files, one at a time, and appends compressed bins to the audit trail file.

The syntax of command **audit** is

```

audit on      file_name
audit off

audit event   one_of(+|-)  event_name  ...
audit sevent  one_of(+|-)  event_name  ...

audit user    one_of(+|-)  user_name    ...
audit group   one_of(+|-)  group_name   ...

audit clear   one_of(event|sevent|user|group)

audit status  one_of(all|state|event|sevent|user|group|trail)

```

5.5. Using the Audit Print Command

The audit print command **auditpr** reads the audit trail defined by its given *path* argument and prints a report in "attribute file format" (see the AIX manual page for the attributes file format in [AIX OS TR], Chapter 4) on the standard output. It can be invoked only by the superuser.

When invoked with no options, command **auditpr** prints all audit records from audit trail file *path*. When invoked with options, command **auditpr** uses the options as a filter to print only the option-selected audit records from audit trail file *path*. Multiple different options (e.g., `-u john -g system`) are AND'ed together in the filter. The selectable options include user name (login, real, effective), event name (base or administrative), node name, and time (before or after).

If any **auditpr** command option is used, then the resulting attribute file has a default stanza that identifies the options, and those name-value pairs with a single value are factored out of the records that follow. Options with multiple values appear as commented-out comma-separated values in the default stanza. Record stanza names begin with an "r" (for record), followed by the relative record number from the *path* parameter, starting from 1. A comment line containing "***" separates an audit record head from an audit record tail.

Example. This example, which prints all records for node with nid 0x20814447, shows some sample output in "attribute file" format. Here, *luid* is the login user ID.

```

auditpr -n 20814447 /audit/traill

* auditpr output from /audit/traill
default:
    nid = 0x20814447

r1:
    event = link
    type = SYSCALL
    luid = 200
    ruid = 0
    euid = 0
    rgid = 0
    egid = 0
    pid = 89
    ppid = 57
    time = "Aug 24 13:28:22 1987"

***
    error = 0
    rval1 = 0

```

```

rval2 = 1073734951
npaths = 2
path1 = x
path2 = y
nargs = 2
arg1 = 1073734947
arg2 = 1073734951

r2:
    ...

```

5.6. Adding Auditing to a Privileged Program

To avoid flooding the audit trail with unnecessary records, it is often desirable to turn off, or temporarily suspend then resume "normal" auditing (e.g., system call events) in a privileged process, and cut only a few selected records. A (trusted) programmer can use system call `auditproc()` in a privileged program to suspend normal auditing, and system call `auditlog()` to cut selected audit records. [P] provides this too.

Interestingly, all auditing system calls are privileged. Even `auditlog()` is privileged to help maintain the integrity of the audit trail file. Thus, audit records cannot be cut from an unprivileged process (e.g., `/bin/sh`, `/bin/tsh`). By the way, `/bin/tsh` is trusted but not privileged.

6. Integrity Enhancements

AIX' has a command to check installation assertions on system security files, and a command to check integrity assertions against system security tables. AIX' uses a file of Prolog-like facts to drive the table checker, and provides a family of subroutines for safely updating (with update-in-place) and reading security tables.

6.1. Checking the Integrity of System Security Tables

A necessary condition for AIX' to be in a secure state is that the system security tables satisfy various intra-table and inter-table integrity assertions (or data invariants). On AIX', the system security tables include:

```

/etc/group
/etc/passwd
/etc/security/s_cmd
/etc/security/s_group
/etc/security/s_install
/etc/security/s_installx
/etc/security/s_user

```

By intentional design on AIX', all these tables have the same format; each is a so-called *table-file*, a file with the same format as the password file `/etc/passwd` and group file `/etc/group`, and a file that consists of entries each with colon-separated fields and newline-separated entries. Because these tables have the same format, we can read and write them with the same set of subroutines, the `tbl` subroutines. To update atomically in-place one of these tables, we can use a common subroutine like `tblsetrec`.

These tables have various integrity assertions that help define a consistent security table state, violation of which may compromise secure system operation. In ordinary system operation, these tables start with a consistent state, and step through new consistent states by atomic updates that preserve state consistency. These integrity assertions can be violated by a system administrator that directly edits one or more of these files, or very infrequently by some system failures. Currently, nothing other than "administrative procedure" prevents a system administrator from directly editing one of these files, which is a blessing sometimes when a system administrator needs to fix one of these files in a way that may not be possible with the commands (e.g., `adduser`) that

manage the files. Many of these assertions are well-known to experienced system administrators, and stated on various manual pages.

However, it is desirable for a system administrator to be able to check these assertions with a command, and it is desirable for possible future extension that the assertions not be compiled into the source code of the command as control flow but instead be represented in a file that drives the command. In other words, we should have a table-driven checker of integrity assertions on system security tables.

Two candidates for such a checker are `sh` and `awk`. Since neither is in the TCB of AIX', and each is possibly too large to add to the TCB without considerable assurance effort, another approach is necessary: the *tiny language approach*.

The approach taken in AIX' was to design a *tiny* specialized assertion language, and write the corresponding tiny scanner/parser/interpreter for it, which would be part of the TCB and would be small enough to entail little assurance effort.

The tiny assertion language contains about a dozen built-in assertions and looks like Prolog facts. To understand the tiny language, it is not necessary to learn anything about Prolog. You can understand the assertions by explained examples, the approach we take below. Prolog facts nicely model unconditional assertions about these tables. If the need arises for conditional assertions, the tiny language and its interpreter can be extended easily to include a semantically-restricted version of Prolog-like rules. A `%` (or `#`) character begins a comment, and a newline ends a comment.

Example. Here are assertions about the password file `/etc/passwd`:

```
% passwd
table(passwd, "/etc/passwd", required, user, 7).
field(passwd, 0, uname, required).
field(passwd, 1, waspassword, optional).
field(passwd, 2, uid, required).
field(passwd, 3, gid, required).
field(passwd, 4, comment, optional).
field(passwd, 5, home, required).
field(passwd, 6, shell, optional).
unique(passwd, uname).
is(passwd, gid, group, gid).
is(passwd, uid, s_user, uid).
directory(passwd, home).
executable(passwd, shell).
```

The first line is a comment. The `table/5` predicate (the `"/5"` notation means that the table predicate has 5 arguments) states that we define a table with the following attributes: its shorthand name in these assertions is `"passwd"`, its absolute path is `"/etc/passwd"`, its existence is required, each entry is called a `"user"`, and each entry has 7 fields. The `field/4` predicate defines attributes of a field of a table. For example, `field 0` of the `passwd` table has a shorthand name of `"uname"` in these assertions, and it must have a nonempty value (its value is required). The `unique/2` predicate here states that the `uname` field values in the `passwd` file are unique, no duplicate user names. The first `is/4` predicate states that the `gid` field value in each `passwd` table entry is also a `gid` entry in the `group` table. The second `is/4` predicate states that the `uid` field value in each `passwd` table entry is also a `uid` entry in the `s_user` table. To complete the example, we would need a `table` predicate for the `group` and `s_user` tables and `field` predicates for the fields. The `directory/2` predicate states that the `home` field of the `passwd` table must be a directory, and the `executable/2` predicate state that the `shell` field of the `passwd` table must be an executable file.

The assertion table drives the security table check command `stableck`. Command `stableck`, for specified or all tables, prints a message for each failed assertion.

6.2. Checking the Installation of Trusted Files

Another necessary condition for AIX' to be in a secure state is that all trusted files are installed correctly. Files `s_install` and `s_installx` are table-files that hold installation knowledge for all trusted files. File `s_install` holds installation knowledge for trusted files in the base AIX' system, and is read-only, whereas file `s_installx` holds installation knowledge for trusted files in the AIX' system *extended* with one or more applications. Together, these tables are a registry of all trusted files in the AIX'. In AIX', command `sinstalck` (read, "install check") checks the installation of trusted files.

Without any arguments, the `sinstalck` command makes two checks; in Step 1 (the "installation check" or "all check") it checks that every entry in an installation table is installed correctly, and in Step 2 (the "tree check" or "only check") it checks that under the root "/" there is no setuid-root program not posted in an installation table. It provides some details on the check progress, and it stops when it finds a problem. In addition, it also permits the system administrator the option to fix certain installation problems.

6.3. Safely Updating System Security Tables

Commands that update system security tables (e.g., `passwd`, `adduser`) should use a common subroutine with well-defined, well-scrutinized semantics for correct updating. In an environment with AIX Distributed Services, update-in-place must be used due to file-over-file vmounts, otherwise the new version may not be in the same virtual file system. We designed, added, and used such a subroutine in AIX'. It allows many readers not to block a writer, allows a slow writer not to block readers, serializes writers, and keeps a table backup copy.

7. Lessons Learned

On configuring security features. (1) Make C2 security features configurable to accommodate several user communities. (2) Disable strict security features in the default (shipped) system to avoid potential UNIX/security culture shock. So, for tighter security, you must explicitly configure the features you want. As an exception in AIX', the SAK is permanently configured. (3) Centralize the enabling and disabling of security features in a single file, say `/etc/security/security.cfg`.

On the superuser. In AIX' in multi-user mode, it is unacceptable for a superuser to execute (especially user-supplied and potentially all) untrusted commands, otherwise it is possible to steal superuser authority. AIX' enforces this regime. When C2 is configured in AIX', you can `su` to root only inside the trusted path, and you cannot escape to (or use) an untrusted shell as superuser. From the trusted path in the trusted shell, only trusted commands can be executed.

On a programming discipline for trusted applications. We learned a programming discipline for writing trusted applications for a design targeted at the class C2 level with a class B3 trusted path mechanism.

On table-driven commands. For flexibility and extensibility, we have had enjoyable experience with the new table-driven commands. It is far easier to revise a specification in a table than it is to change, recompile, and reinstall code. We do need more experience identifying and measuring performance issues, though. Table-driven commands are well suited to both experimental and operational environments. Certain design decisions, not just configurable parameters, are exposed and consolidated, not buried in compiled code. Rather than complicate system administration, they tend to unify it.

On a trusted path. Having a B3 trusted path in a C2 system simplifies the overall assurance effort of a trusted system. The trusted path, together with a table-driven trusted shell, allows our design to satisfy easily other security requirements (e.g., trusted importing of "unlabeled" data) with no additional mechanism.

8. Summary and Conclusion

In summary, AIX' contains the following features:

- auditing subsystem
- tighter I&A (login user ID, no pseudo-user login, configurable password checks)
- trusted path (SAK, SIGSAK, trusted shell, command table)
- hardcopy labeling
- integrity enhancements (table check, installation check, safe table updating)
- TCB scrutiny (auditing and "trustedness" changes to trusted programs)

By prototype demonstration, we conclude that configurable, C2 security features can be integrated into the base UNIX operating system. In other words, for C2 security we can have a "UNIX with the Superuser" [H1].

Acknowledgments

We acknowledge the following people for helpful discussions on the ideas in this paper: Sekar Chandrasekaran, Prof. Virgil Gligor, Doug Steves, Ken Witte, Grover Neuman, N. Vasudevan, Mark Carson, Wilhelm Burger, Sohail Malik, Chii-Ren Tsai, Gary Luckenbaugh, Jack O'Quin, Nick Camillone, Shau-Ping Lo, and Janet Cugini. We also acknowledge the following people for supporting our efforts: at IBM SID, Gerry Ebker, Walt Harner, Bob Crago, Bob Arthur; and at IBM ESD, Bill Sandve, Khoa Nguyen, and Bob Willcox.

References

- [AIX OS CR] *IBM RT PC AIX Operating System Commands Reference* (1988).
- [AIX OS TR] *IBM RT PC AIX Operating System Technical Reference* (1988).
- [Bi] Bishop, M., "How To Write a Setuid Program," *:login:*, the USENIX Association Newsletter, Vol. 12, No. 1, pp. 5-11 (January/February 1987).
- [Bu] Bunch, S., "The Setuid Feature in UNIX and Security," *NBS/NCSC 10th National Computer Security Conference Proceedings*, pp. 245-253 (September 1987).
- [G+] Gligor, V.D., *et al.*, "Design and Implementation of Secure XENIX," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 2, pp. 208-221 (February 1987).
- [H1] Hecht, M.S., *et al.*, "UNIX without the Superuser," *1987 Summer USENIX Conference*, Phoenix, Arizona (June 1987).
- [H2] Hecht, M.S., *et al.*, "How To Write a Trusted Application," unpublished manuscript (1987).
- [L] Lennert, D., "How To Write a UNIX Daemon," *:login:*, the USENIX Association Newsletter, Vol. 12, No. 4, pp. 17-23 (July/August 1987).
- [P] Picciotto, J., "The Design of an Effective Auditing Subsystem," *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, California, pp. 13-22 (April 1987).
- [PMG] Department of Defense Computer Security Center, *Department of Defense Password Management Guideline*, CSC-STD-002-85, Library No. S-226,994 (April 12, 1985).
- [S+] Sauer, C.H., Johnson, D.W., Loucks, L.K., Shaheen-Gouda, A.A., and Smith, T.A., "RT PC Distributed Services Overview," *Operating Systems Review*, Vol. 21, No. 3, pp. 18-29 (July 1987).
- [TCSEC] Department of Defense Computer Security Center, *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD (December 1985), which supersedes CSC-STD-001-83, Library No. S225,711 dated August 15, 1983. This reference is commonly called the *Orange Book* after its cover color.

γ -GLA

A Generator for Lexical Analyzers That Programmers Can Use

Robert W. Gray

Computer Science Dept 430
University of Colorado
Boulder, CO. 80309-0430
bob@boulder.COLORADO.EDU

ABSTRACT

Writing an efficient lexical analyzer for even a simple language is not a trivial task, and should not be done by hand. We describe GLA, a tool that generates very efficient scanners. These scanners do not use the conventional transition matrix, but instead use a few 128 element vectors. Scanning time is only slightly greater than the absolute minimum — the time it takes to look at each character in a file. The GLA language allows simple, concise specification of scanners. Augmenting regular expressions with *auxiliary scanners* easily handles nasty problems such as C comments and C literal constants. We formalize the connection between token scanning and token processing by associating a *processor* with appropriate patterns. A library of canned descriptions simplifies the specification of commonly used language pieces — such as, C_IDENTIFIERS, C_STRINGS, PASCAL_COMMENTS, etc. Finally, carefully tuned lexical analysis support modules are provided for error handling, input buffering, storing identifiers in hash tables and manipulating denotations.

1. INTRODUCTION

Of the hundreds of 4.3BSD software applications that perform lexical analysis, only a couple of them use the *lex* tool to generate their scanners. The "serious" applications such as *cc*, *as*, and *c++* all have hand written, carefully tuned lexers. We hypothesize that this is due to efficiency considerations and inappropriate specification language.

This paper attempts to convince you that:

- (1) Writing an *efficient* and *complete* lexer for simple languages is not trivial.
- (2) A tool generated lexer can match, and even exceed the speed of hand written lexers — even the carefully tuned ones.
- (3) There are better languages than *lex* for specifying lexical analysis. A more compact and concise specification is desirable and possible.
- (4) There is a common set of support modules that most lexical analysis tasks require. By standardizing the interface to these routines and placing them in a library, the chore of rewriting them each time is avoided.

In response to Point 1, Aho, Sethi and Ullman [Aho1986] give a method for implementing transition diagrams. Granted, the resulting program is simple. However, it handles only a pure finite state machine, is not efficient, ignores lexical error handling, and defers the problem of what to do with identifiers, strings and integers after they have been scanned. Dealing with all of these issues presents a programmer with a lot of work — even for a trivial language. We would rather let a tool handle all of the work of building an efficient scanner and providing the

related support modules. With γ -GLA¹, a straightforward compact specification yields an efficient scanner and its comprehensive lexical analysis environment.

Point 2 deals with efficiency. First some measurements will illustrate the importance of efficient lexical analysis. Using *gprof* [Graham1982], I measured the time spent lexing for two heavily used programs — the portable C compiler, and the assembler (4.3BSD). Both of these use hand written, carefully tuned lexical analyzers. A reasonably "typical" input file, *as/assyms.c*, was used. More than 40% of assembly time was spent performing lexical analysis! About 10% of the time was just in the scanner² itself. For the C compiler, which does a lot more work than the assembler, almost 20% of the total time is spent doing lexical analysis. The scanner was the 4th most expensive routine!

According to Jacobson [1987], (an author of *flex*), a *lex* generated scanner can be tuned to gain a factor of 10 performance improvement. Assuming this premise and the *gprof* data, if an inefficient tool generated lexer were dropped into the assembler, the overall time could double. With the compiler, the numbers are not so dramatic, yet still significant. At the 1986 Atlanta USENIX conference, Honeyman [1986] noted that *pathalias* sped up a factor of two when its *lex* generated scanner was replaced by a hand-coded scanner.

Efficiency does not have to be compromised to use a tool generated scanner. In fact a GLA generated scanner runs only slightly slower than the following loop!

```
while ( (c = getchar()) != EOF) /* getchar is stdio.h macro */
    ; /* empty */
```

This loop is the simplest example of a program that "touches" every character — as a scanner also must do. Clearly, an important aspect of scanning is efficient input buffering — as we will see, a possible pitfall of hand written lexical analyzers.

The hand written scanner of the C compiler has missed some optimizations. It uses the standard I/O buffering which is about twice as slow as the sentinel method [Aho1986]. Figure 3 confirms this. The C compiler uses non-compact, and nested case statements, which costs up to 25% more time (Waite [1986a]). When benchmarked against a GLA generated scanner, the hand written C scanner was about 30% slower.

Point 3 deals with the human interface aspects of lexical analysis. A *specification* of what some code ought to do should be significantly clearer and more transparent than the *code* itself. For example, the lexer for the C compiler is almost 1000 lines of fairly tricky, tuned code. It would take a long time of looking at the code to be able to precisely describe the behavior of that lexer. On the other hand, it is possible to specify the lexical aspects of the C language in 100-200 lines of *lex* code and in 50-100 lines of GLA code of which most are automatically extracted from the parsing grammar. Ideally, all lexical specifications will be short and concise. Now, the problem is to take a clear specification, and generate efficient code from it. The *lex* language is not best for the task. For example, Schreiner and Friedman [Schreiner1985] propose the following *lex* pattern to recognize a C-style comment:

```
"/**"/"*/"([^\n/])["/*"]"/"*/"([^\n/])/*""**"/"
```

Simpler patterns might not recognize */** as beginning a comment or */**/* as being a comment. Clearly, regular expressions by themselves do not lead to transparent specifications of certain

¹ Although the first two versions — α -GLA and β -GLA produced high speed scanners, their specification language proved to be too constraining, hence the need for the γ version. For the remainder of this paper, the current version will be referred to as GLA.

² In this paper, *scanning* is the task of partitioning input into pieces (tokens). *Lexical analysis* includes scanning, input buffering, error handling and token processing — such as inserting an identifier into a hash table or converting an ascii integer to binary. *Lexing* is an abbreviation for performing lexical analysis.

language constructs. *lex* has the notion of "context conditions" that helps the above problem, but these cause a simple specification to begin to look like "spaghetti" code (Figure 4 has a trivial example of context conditions for Pascal comments). Furthermore, as we will see, "context conditions" are not the most efficient way to scan tokens such as strings and comments.

Consider the task of specifying the tokens of a scanner language itself. The *lex* scanner requires more than 500 lines of C code. *flex* [Paxson1988], which has approximately the same language as *lex*, uses 378 lines to describe its basic symbols. Most of these lines are patterns and actions. There are 18 context conditions which make the specification non-trivial to understand. Although this specification is an improvement over the original hand coded *lex* scanner, there is still lots of room for improvement. GLA specifies itself in 19 lines of which 15 lines were automatically generated. In other words, there are 4 lines to describe Identifiers, Integers, Regular Expressions and comments of the language.

GLA achieves this conciseness of specification by incorporating several features:

- GLA formalizes the connection of auxiliary scanners to regular expressions, that is, the initial part of certain tokens is easily described by a regular expression. An auxiliary scanner then takes over to specialize in recognizing the rest of the token. A good example of this is multi-line C literal constants. Another example is comments, either nested or non-nested.
- Most non-literal tokens (identifiers, integers, strings, etc.) require processing after they have been scanned. The specification describes what *processor* to invoke after such a token is scanned. Most common processors are already provided in a GLA library.
- GLA provides a shorthand notation called "canned descriptions." By mentioning a keyword in the specification, the appropriate regular expression, auxiliary scanner and processor for a non-literal token are substituted. For example, a programmer can build a lexical analyzer for a language by asking for, C_IDENTIFIERS, ADA_COMMENTS, and MODULA_INTEGERS.

Point 4 maintains that certain tasks have to be performed in almost every lexical analysis situation and there is no reason to have programmers rewrite these routines for every application. There are many pitfalls to writing a good identifier table module. It is not obvious how to write very efficient memory allocation for string storage. Normalizing floating point numbers and checking for overflow are crucial for avoiding compiler crashes when folding constants, but again these take significant time to write.

GLA is a lexical analysis generating tool with a very powerful and concise specification language. It produces lexers that are as fast or faster than carefully tuned, hand written ones. Section 2 discusses how the efficiency is obtained. Section 3 describes the specification language. Section 4 describes the very important support modules that are paramount to an overall efficient lexical analyzer.

2. EFFICIENCY OF SCANNERS

Section 2.1 discusses the conventional model of scanning and Section 2.2, the GLA model. The performance of the two methods are compared in Section 2.3.

2.1. Conventional Scanners

The model that most scanners follow is the finite state machine (FSM). A FSM operates by examining the current state and the current input character to decide what to do next. The transition function is represented by a matrix, `table[state,character]`, which is interpreted during scanning. The following code outlines the process:

```

while (basic symbol not complete)
    state = table[state, character]
    switch (state)
        case 0: { code to be performed in each state}
        ...
        case i:
        ...
        case laststate:

```

Notice that the fundamental decision being made at every loop iteration is

What state do I go to next, based on current state and character?

The main issue with conventional scanners has been how to compact the large, sparse matrix. Pascal, a relatively small language, has 165 states, and if 7 bit ASCII is the character set, requires a matrix of size 21,120. Any table compression scheme will have to trade off speed against size. *lex* tries to optimize space at the cost of time, a holdover from PDP-11 days. Furthermore, the main recognition loop of a *lex* scanner is slowed down by the cumulative cost of several infrequently used features. *Flex*, which is a rewrite of *lex*, saves the user from paying for unused features and also allows the user to choose the time/space tradeoffs for the transition matrix compression. As we will see, even with all of these improvements, *flex* generated scanners will be twice as slow as GLA generated scanners.

As a side note, a commonly used idea to reduce the number of states and consequently the table size is to treat keywords as identifiers during scanning, then a search is made in a hash table to determine if the scanned token is a keyword or an identifier. For Pascal the number of states drop from 165 to about 37 when keywords are scanned as identifiers. Both Aho [1986] and Schreiner [1985] give details for this approach. It is also discussed further in Section 3.6.

2.2. Directly Executable Scanners

Figure 1 shows a fragment of the GLA produced scanner for recognizing the following simple patterns (the last two are taken literally):

```

[0-9]+
[A-Z][A-Z0-9]*
<=
◇

```

Whereas a conventional scanner does a table access and "switches" on every input character, the directly executable scanner:

- (1) In the ideal case, spins in a tight "while" loop as shown below (note that the "while" loop executes the empty statement — ;). These loops correspond to seeing the second through the last characters of identifiers (and keywords), numbers, strings, whitespace, etc. Any place in the FSM where a transition goes from a state to itself, the ideal case occurs.
- (2) Otherwise, a typically short "if" cascade is used. The transition is implemented with a "goto" which is faster than a "switch".

CaseTbl as shown in Figure 2, is a 128 element array that "drops" us into the directly executable code based on the first character of a token.

```

switch(tokType = CaseTbl[c = *p++]) {
    case 1:  /* 0-9 was seen as first character of this token */
        /*2*/ while(scanTbl[c= *p++] & 1<<0) ; --p; /* more digits */
              TokenEnd=p; /*make progress after scan*/
              goto State_other;

    case 2:  /* A-Z was seen as first character of this token */
        /*4*/ while(scanTbl[c= *p++] & 1<<1) ; --p; /* 0-9A-Z */
              TokenEnd=p; /*make progress after scan*/
              goto State_other;

    case 3:  /* < was seen as first character of this token */
        /*3*/ /* not a final state */
              if( (c = *p++) == '=' ) goto St_5;
              else if( c == '>' ) goto St_6;
              else { --p; goto State_other; }

    St_5:  TokenEnd=p; /*tokType=3; token is <= */
          goto State_other;

    St_6:  TokenEnd=p; tokType=4; /* token is <> */
          goto State_other;

    ...

```

Figure 1. Directly executable scanner

```

static short CaseTbl[128] = {
    ...
    1, 1, 1, 1, 1, 1, 0, 0, /* 4 5 6 7 8 9 : ; */
    3, 0, 0, 0, 0, 2, 2, 2, /* < = > ? @ A B C */
    ...

    static char ScanTbl[128] = {
        ...
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x00, 0x00, /* 4 5 6 7 8 9 : ; */
        0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x02, 0x02, /* < = > ? @ A B C */
        ...

```

Figure 2. CaseTbl and ScanTbl

The tokType is set in the switch to the most probable token, but can be reset (as is necessary for St_6 but not necessary for St_5).

ScanTbl is a collection of 128 element bit vectors (up to eight for "char" data type, only 2 used in this example) indexed by the current input character. It is used to quickly recognize characters belonging to certain classes. For example, the zeroth bit represents the class [0-9] and the oneth bit, the class [A-Z0-9].

2.3. Speed comparison

The time³ required by various programs to scan about 10,000 lines of Pascal code is given in Figure 3.

Program	Time			Space			
	user	sys	total	text	data	bss	total
gla	1.4	0.2	1.6	16384	8192	6272	30848
flexcf	3.18	0.24	3.42	16384	32768	2696	51848
flexcem	7.48	0.36	7.84	16384	8192	7008	31584
lex	11.92	0.18	12.1	16384	8192	12820	37396
getchar	0.92	0.1	1.02	8192	8192	1884	18268
source	0.22	0.2	0.42	16384	8192	5008	29584

Figure 3. Time and space requirements of scanners

The 317,439 byte file is three copies of the SYNPUT program. Detailed characteristics of this program are given by Waite [1986a] but briefly, the most frequent tokens are: 17,397 Identifiers with average length of 3.9 characters, 8,289 single spaces, 6,102 keywords with average length of 3.7 characters, 5,640 semicolons, and 5,511 occurrences of 3 or more consecutive spaces, each occurrence averaging 10.8 spaces.

The GLA scanner runs at maximum speed when scanning successive characters of tokens such as identifiers, keywords, numbers and comments. The trend towards longer identifiers, and more comments in programs takes advantage of the maximum speed of GLA scanners.

The times are averaged for 5 trials on an unloaded Sun 3/260 with its own local disk. A different copy of the data was used for each trial to avoid any disk cache "hits". All programs were compiled with the -O optimize flag. The *time* command provided the speed measurements. The *size* command provided the space information for *text*, (the executable code), *data*, (the initialized data), and *bss* (the uninitialized data, zero fill on demand). (The link editor rounds up sizes to the next 2k byte page boundary).

The body of the *getchar* program was given earlier in the introduction. The program *source*, does the equivalent task, but instead of using Standard I/O, it uses our own buffering. The factor of two speedup is due to using an ascii NUL sentinel at the end of a buffer, rather than explicitly maintaining a character count which requires an extra decrement and test instruction.

The suffixes on the names of the flex programs are compression options, from least compressed, but fastest to most compressed but slowest. For example, the program flexcf was obtained by

```
flex -cf
```

which means, produce a scanner with full tables (no compression). The -cem option optimizes space — as is reflected in the total size.

The gla scanner is twice as fast as the fastest flex scanner, yet it is still smaller than the most compressed flex scanner.

³ Note, this data is in contradiction to Jacobson's statement "When tested against the hand-coded scanners described in [Waite] and [Heuring] the new Lex was always faster." Since Jacobson's paper was never published, the discrepancy has not been resolved.

For the measurements, both flex and lex used the specification given in Figure 4. The GLA specification (processors were not invoked for the measurements) is given in Figure 6.

Based on the performance data, and studying the source code of both lex and flex, I have formed the opinion that flex makes lex obsolete. Even though GLA is clearly superior in performance to flex there are cases where flex is still needed. If a specification has been written in lex, substantial performance improvement can be gained with little effort by switching to flex. GLA would require a new specification. Also flex has a couple of features such as "interactive" input and YYREJECT, that were not a part of the GLA design goals.

We can gain some insight into how the GLA scanner obtains its high speed by looking at the four 68000 instructions generated for one of GLA's tight "while" loops.

```
while(scantbl[c = *p++] & 1<<2)
```

```
;
```

L68:	movb a4@+,d0	Get an input character and incr pointer
	extw d0	Make it a suitable index
	bst #2,a5@(0,d0:w)	Check whether it is in the class
	jne L68	If so, continue while loop

Notice the use of hardware registers a4, a5, and d0 for the C variables register char *p, the base of scanTbl, and register int c respectively. The mask (here "#2") defines a particular bit vector

```
%s COMMENT
%%
"(" BEGIN COMMENT;
<COMMENT>"*" BEGIN 0;
<COMMENT>[~*\n]+ {linenum++; }
<COMMENT>\n {linenum++; }
[0-9]+ {return(2); }
[A-Za-z][A-Za-z0-9]* {return(3); }
[0-9]+\ [0-9]+ {return(4); }

"(" {return( 6); }
"." {return( 7); }

...
\t {tabs++; }
\040 {blanks++; }
\n {freelines++;}
. { /*ECHO;*/ others++; }
```

Figure 4. Flex and lex specification for Pascal

of scanTbl as discussed above.

The conventional finite-state machine model will generate the following longer and more expensive sequence:

```

do {   state = table[state, *p++]
      } while (state > 0)

L40:  movb  a5@+,d0           Get an input character and incr pointer
      extw  d0               Make it a suitable index
      movl  a4@(0,d0:w:4),a0   Get the state table column
      movb  a0@(0,d7:1),d7     Get the next state
      jgt   L40              Continue if nothing special

```

We see no way to make the conventional model go faster — it is doing more work by asking the more general fundamental question:

What state should I go to next?

The GLA scanner asks the easier question:

Should I stay in the same state?

This latter question seems to be powerful enough for the task of scanning programming languages.

3. USER INTERFACE

Section 3.1 presents the GLA grammar and highlights the major departures from *lex*. Sections 3.2 and 3.3 discuss *AuxScanner* and *Processors*. Section 3.4 explains the convenience of "canned descriptions". Section 3.5 deals with interfacing scanning to parsing. Section 3.6 explains how keywords are preloaded into a table.

3.1. GLA grammar

Figure 5 gives the GLA grammar (brackets indicate optional items).

```

specification:  ((Label ":") nonLitDesc)*
                options+
                ((Label | RegularExpr) Int)*
nonLitDesc:    (RegularExpr | LibrRef) ["(" AuxScanner ")"] ["(" Processor ")"]
options:       "%%" | "%EOFTOKEN" Int | "%MAXCODES" Int

```

Figure 5. GLA parsing grammar

A specification consists of three parts: The non-literal description part, options and the encoding part. In most cases, only the non-literal description part needs to be provided by the user; the other parts can be automatically produced from the parsing grammar (see Section 3.5). *Label*, *AuxScanner*, *Processor* and *LibrRef* are C-style identifiers. *Int* is a decimal integer. A

RegularExpr is a sequence of visible characters delimited on the left with \$ and on the right with whitespace. In the non-literal description part, the meta characters are the same as lex meta characters. The \$ on the left has no meaning in the expression and since no regular expression ever starts with \$ this choice allows us to distinguish between *RegularExpr* and *LibrRef*. In the token encoding part (third part of the specification), the regular expressions are taken literally — that is, there are no special characters. In a GLA specification, comments are enclosed in { }.

Figure 6 shows the interesting part of the Pascal specification (the first five lines are the hand written part, the rest is automatically generated from the parsing grammar):

```

$ \{ \N*      (pascalCommentAuxScan) { no label for comment }
Int_number:   ${0-9}+      [mkint]
Name:         ${A-Za-z}[A-Za-z0-9]*  [mkidn]
Real_number:  ${0-9}+\.[0-9]+      [mkfpt]
String:       $'          (pascalStrScan)  [mkstr]
%%
Int_number    2      { token encodings }
Name          3
Real_number   4
String        5
$(            6
$.           7
...

```

Figure 6. A partial GLA specification of Pascal

The integers are the values returned when the tokens are recognized. For the comment description, there is no label or corresponding token encoding part. In this situation, the scanner will not return to its caller after a comment has been recognized, instead it will continue scanning for the next token. Eventually, a token will be found for which there is a return code, or the end of file code will be returned (default is zero).

String is a good example of *auxiliary scanner* and *processor*. First, the FSM recognizes a single quote. Then the auxiliary scanner *pascalStrScan* is invoked with a pointer to what has been recognized and its length. When scanning is complete, *mkstr* is invoked to store the string and return a handle to it. *pascalStrScan* is a routine that has been optimized to scan Pascal strings, and handling the embedded quote problem. Note that the simple regular expression

`$'.*'`

will not work for Pascal strings because in the example

```
writeln( 'the ' count ' most frequent scores average ' ave)
```

the pattern would consume both strings and the embedded variable *count*.

3.2. AuxScanner

There are cases where it is very convenient to use auxiliary scanners, such as for recognizing comments and strings. A typical *nonLitDesc* using an *AuxScanner* would be:

```
$"/*"      (CcommentAuxScanner)
```

For illustrative purposes, Figure 7 gives the body for the auxiliary scanner of C-style comments. This is one of many auxiliary scanners that have already been written and are provided in the support library. After the main GLA scanner recognizes the *RegularExpr*, (/* in this case), *AuxScanner* is invoked as

```
p = AuxScanner(TokenStart, TokenEnd-TokenStart);
```

where the first parameter is the start and the second parameter gives the length of what has been recognized by the FSM.

This example serves as a template for all auxiliary scanners — the bookkeeping associated with tabs, newlines and buffering is the same. The expression in the "while" loop can easily be optimized into a bit vector as explained earlier. In the frequent case, this would reduce the per character testing from four tests to one test.

```
top: while( bitvector[c = *p++] ) /* null statement */;
```

bitvector should be non-zero for all but newline, tab, star and nul.

```
char * CcommentAuxScan(start, length)
char * start; /* start of characters recognized by reg expr */
int length; /* length of what was recognized in reg expr */
{ register char c;
  register char *p = start + length; /* first char not yet processed */

  top: while( (c = *p++) && c != '*' && c != '\n' && c != '\t' )
    ;
  if( c == '*' ) {
    if( *p == '/' ) return(++p); /* we have a comment */
    goto top; /* not a comment ending */
  }
  else if( c == '\n' ) { /* bookkeeping */
    LineNum++; StartLine = p-1; goto top;
  }
  else if( c == '\t' ) { /* bookkeeping */
    StartLine -= 8-(p-StartLine-1)%8; goto top;
  }
  else /*if( c == '\0' )*/ { /* buffering */
    refillBuf(p-1); p = chpt;
    if ( *p == '\0' ) message(ERROR, "file ends in comment", 0, &curpos);
    else goto top;
  }
}
```

Figure 7. C-style comment auxiliary scanner

3.3. Processors

Processors are routines that are invoked after recognizing tokens with intrinsic values. For example after an identifier has been scanned, we usually wish to associate a small integer with its string representation. This is often accomplished with a hash table which is just what the processor *mkidn* does. The small integer is stored by the processor at a location specified by the caller (*value* below). After the characters of an integer are scanned we usually wish to know the value of the integer (also stored in *value*). The processor *mkint* handles this. The library of processors also contains routines for saving strings, normalizing floating point numbers, etc. (See the GLA Reference Manual [Gray1988]). Processors are invoked as:

```
Processor(start, length, &tokType, &value)
```

The reason for the address of *tokType* is that the processor has the option of overriding the decision of what token type is to be returned for the scanned characters. This makes sense because a processor is more specialized than a FSM operating with a regular expression. The most common case comes up handling keywords. Keywords can be "preloaded" into the identifier table (Section 3.6). In most languages, keywords are "matched" by the pattern for identifiers. The processor *mkidn* will attempt to load a keyword into the table as an identifier, but *mkidn* will know that it is not an identifier, but instead a keyword, and will set the *tokType* appropriately. For example, given the following line of C code and given that "initProcess" was recognized by the scanner:

```
initProcess = (int *) malloc (...);
```

then the processor, *mkidn* would be called with *start* pointing to the beginning of the token "initProcess", length set to 11, and *tokType* == IDENTIFIER. Upon return, "initProcess" would be installed in the identifier table, and *value* would point to it.

On the other hand, after *mkidn* processes "int" above, the *tokType* would be reset to KEYWORD_INT and *value* would be irrelevant.

3.4. Library of non-literal descriptions

The task of specifying non-literals can be tricky and error prone. We believe that the problem can be minimized by providing what programmers need for the majority of the cases. There are two advantages of "canned" descriptions: 1) most of the time, the descriptions can be used unchanged — this contributes to simplicity and reliability, 2) for the infrequent cases, it is easier to modify something close to what is needed rather than starting from scratch.

For example, the canned description for PASCAL_IDENTIFIER is:

```
$(a-zA-Z)(a-zA-Z0-9)*      [mkidn]
```

If one had a requirement to allow \$ in their identifiers, only a trivial addition needs to be made. New languages can easily be built from pieces of other languages. For example, the following GLA specification, coupled with the literal extraction of the next section is all that is needed for the lexer description.

```
idn:  C_IDENTIFIER
      ADA_COMMENT
int:  MODULA_INTEGER
str:  PASCAL_STRING
```

3.5. Interfacing scanners to YACC parsers

The lex manual suggests interfacing Yacc and lex in the following manner. Lex writes a program named *yyllex()* which Yacc will call when tokens are needed. Lex rules should end

with

```
return(token);
```

An easy way to get access to Yacc's names for tokens is to compile the lex output file as part of the Yacc output file by placing

```
#include "lex.yy.c"
```

in the last section of Yacc input.

The problem with this approach is that it still leaves a lot of tedious, mechanical (and error prone) work for the programmer. First the programmer is required to invent names for literal tokens. For example if you desire the following grammar:

```
variable ":=" expr
```

you need to write it as

```
variable ASSIGNOP expr
```

Next, ASSIGNOP needs to be listed in the *token* section of the grammar. Finally, later in the lexical specification, there is the need to reassociate the two:

```
":=" {return(ASSIGNOP); }
```

This has to be done for every literal — the small Pascal language has about 60 different literals.

We provide the program *litextr* to eliminate this chore. The parsing grammar is written with the literal strings embedded in the rules. *litextr*, given this grammar, Figure 8 (a) and a GLA description of the non-literal tokens (b) produces a Yacc acceptable grammar and a GLA specification that has been completed with literal tokens and encodings for all tokens.

```

                                %token Int Idn
                                %%
                                expr  :    expr "<=" expr
                                      |    expr "<"  expr
                                      |    expr "MOD" expr
                                      |    Idn
                                      |    Int ;

a) Grammar with literals

                                Int:    C_INTEGER
                                Idn:    C_IDENTIFIER

b) Non-literal description

```

Figure 8. Parsing grammar and non-literal description

Figure 9 gives the results of *litextr*: a complete Yacc specification and a complete GLA specification.

3.6. Preloading keywords

For typical languages, there is a huge reduction in the size of the FSM when both keywords and identifiers are scanned by the same pattern. After scanning, a processor can determine if the characters form an identifier or a keyword. A handy implementation of this scheme is to *preload* keywords into a hash table that is also used to handle identifiers. The program *litextr*

```

%token Int          1
%token Idn          2
%token t3_LE        3
%token t4_LT        4
%token t5_MOD        5
%%
expr : expr t3_LE expr
     | expr t4_LT expr
     | expr t5_MOD expr
     | Idn
     | Int ;

```

a) Parsing grammar for Yacc

```

Int:    C_INTEGER
Idn:    C_IDENTIFIER
%%
Int     1
Idn     2
$<=    3
$<     4
$MOD    5

```

b) Complete GLA specification

Figure 9. Complete Yacc parsing grammar and GLA specification

collects the keywords and hands them to *adtinit*, which loads them into a hash table. Then *adtinit*, dumps the hash table in a form that can later be compiled with the rest of the lexical scanner. This scheme yields very fast run time recognition, with no overhead of reading keywords and building the hash table at initialization time.

4. SUPPORT MODULES FOR LEXICAL ANALYSIS

A writer of a lexical analyzer will often wish to use the services of five classes of support modules. We believe that the provided library handles most typical needs; otherwise modules can be tailored or new modules added. The support library consists of:

- (1) **Source text input.** This module provides the input text buffering in the most efficient manner possible. Basically it gives the scanner access to entire lines of input, and never splits lines on buffer boundaries. The two main routines are *initBuf* and *refillBuf*.
- (2) **Error reporting and source text output.** This module provides a standard error reporting mechanism that is invoked whenever an error is detected. GLA generated scanners make available the line and column of tokens recognized. These values can easily be retained by a parser or semantic checker to later exactly position messages in the original source text. The routine *message* takes a error string and coordinates and writes this information in a form suitable for *lisedit* or *error(1)* which disperses the diagnostic error messages to the source file and line where the errors occurred.

- (3) **Character string memory.** This module is used to store the text of identifiers and keywords, and other character strings. Its implementation is considerably more efficient than malloc.
- (4) **Identifier table.** This module implements the unique representation of identifiers and keywords. It is invoked by the scanner to provide the additional information that characterizes the particular identifier or keyword recognized. The identifier table module can be preloaded with keywords.
- (5) **Denotation value management.** This is actually a collection of modules, one for each primitive data type of the source language. They are invoked by the scanner to provide the additional information that characterizes the particular denotation recognized.

5. CONCLUSIONS

A tool that allows one to build software components from a concise specification contributes to reliability and maintainability. Lexical analyzers should be generated by a tool. The language of the tool must allow for a concise, transparent description of the scanner. The GLA model of scanning yields faster scanners than any other general purpose method. Carefully coded support modules are essential for a comprehensive lexical analysis environment and, if provided in a library do not have to be rewritten for each application.

6. ACKNOWLEDGMENTS

Much of this work is based on the work by Heuring [1986] and Waite [1986]. This work was supported in part by the Army Research Office DAAL 03-86-K-0100, and the Office of Naval Research N00014-86-K-0204.

7. REFERENCES

- [Aho1986] Aho, A. V., R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley, 1986.
- [Graham1982] Graham, S. L., P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", *UNIX PROGRAMMER'S MANUAL 4.2BSD, Vol 2C*, 1982.
- [Gray1988] Gray, R., *GLA Reference Manual*, Computer Science Dept, University of Colorado, Boulder, CO, Apr. 1988.
- [Heuring1986] Heuring, V., "Automatic Generation of Fast Lexical Analyzers", *Software—Practice & Experience* 16, 9 (Sep. 1986), ECE Dept University of Colorado.
- [Honeyman1986] Honeyman, P., "PATHALIAS", *Summer USENIX*, Atlanta, JUNE 1986.
- [Jacobson1987] Jacobson, V., "Tuning UNIX Lex (Only abstract)", *Winter USENIX*, Washington, D.C., 1987.
- [Paxson1988] Paxson, V., *FLEX (1) users's manual*, Feb. 1988.
- [Schreiner1985] Schreiner, A. T. and H. G. Friedman, *Introduction to Compiler Construction with Unix*, Prentice-Hall, 1985.
- [Waite1986a] Waite, W. M., "The Cost of Lexical Analysis", *Software—Practice & Experience* 16, 5 (May 1986).
- [Waite1986b] Waite, W. M., V. Heuring and R. W. Gray, *GLA - A Generator for Lexical Analyzers*, ECE Dept University of Colorado SEG-86-1, 1986. 4.

Saber-C

An Interpreter-based Programming Environment for the C Language

Stephen Kaufer, Russell Lopez, and Sessa Pratap
Saber Software, Inc.
saber@harvard.harvard.edu

ABSTRACT

We describe a programming environment that supports an interpreter-based development scheme for the C language. The interpreter contains a parser which loads C files and performs static error checking, an evaluator which executes the intermediate code produced by the parser and performs run-time program checking, a debugger that provides source language debugging, and a linker that supports dynamic linking and incremental relinking of source and object code files. Our goal was to develop an integrated run-time environment for C that promotes prototyping and modular programming, provides comprehensive static and dynamic error detection, and automates the repetitive tasks associated with the development cycle.

1. Introduction

Programming in the C language can be a bittersweet experience. The language's terse syntax and low-level functionality help programmers produce compact, efficient programs that are "close to the machine", earning C the reputation of being a high-level assembly language. However, these same features also invite bugs that evade static debugging techniques and frustrate the most seasoned software developers. Because of the difficulty of locating bugs with existing programming support tools, it is not uncommon for programmers to abandon their debuggers in favor of brute force debugging with *printf* statements.

Attempts to increase the productivity of C programmers have yielded tools that provide better static error detection (*lint*), and compilers that offer better support for run-time error detection [5]. As a further step, a programming environment that integrates the functions of an editor, compiler, and debugger has been implemented to support incremental compilation and better detection of programming errors [6].

These tools adhere to the compiler-based development model for the C language. Source code files are individually compiled to produce object code files which are then linked together to produce a standalone executable program. This development scheme places two limitations on the productivity of the programmer. First, it restricts prototyping and modular development because a fully-linked program is required to execute any component of the application. Second, it places the full responsibility for static and dynamic error detection on the compiler, a demand that cannot be met adequately because of C's loose type requirements for data and pointers.

We have designed a programming environment that supports an interpreter-based development scheme for the C language. Interpreters for the C language have been attempted previously; however, these endeavors either viewed the interpreter as a standalone tool in the UNIX environment [4] or as an extension to the debugger to allow the mixture of interpreted code with compiled code during debugging [2].

Our goal was to construct a single tool that integrated the facilities for creating, testing, and debugging C programs. An interpreter-based development approach was selected to avoid the limitations faced by compiler-based schemes. The following capabilities are offered:

Interactive Run-time Workspace:

A run-time environment is provided to support testing of code fragments, subsections of programs,

and arbitrary C expressions.

Static Error Checking:

Static errors are detected and reported as source files are loaded or source code is entered in the workspace.

Dynamic Error Checking:

Dynamic errors are detected and reported when a program is executed. The checks are designed to detect subtle violations such as errors resulting from pointer misuse or inappropriate references to memory.

Source-language Debugging:

A complete set of debugging tools is provided which can access and modify all source code information contained in the program, including macros.

Fast Turnaround Times:

Changes to source code files are reloaded incrementally; and relinking is only needed for modified modules, not the entire program.

2. Overview of Saber-C

Saber-C consists of a C interpreter, integrated with a dynamic linker, a source language debugger, an editing facility that works with *vi* or *emacs*, and an interface manager. Saber-C runs under the UNIX operating system on Sun and DEC Vax¹ computers equipped with bit-mapped consoles or ascii terminals. The software consists of approximately 120,000 lines of C source code and 50 lines of assembly code.

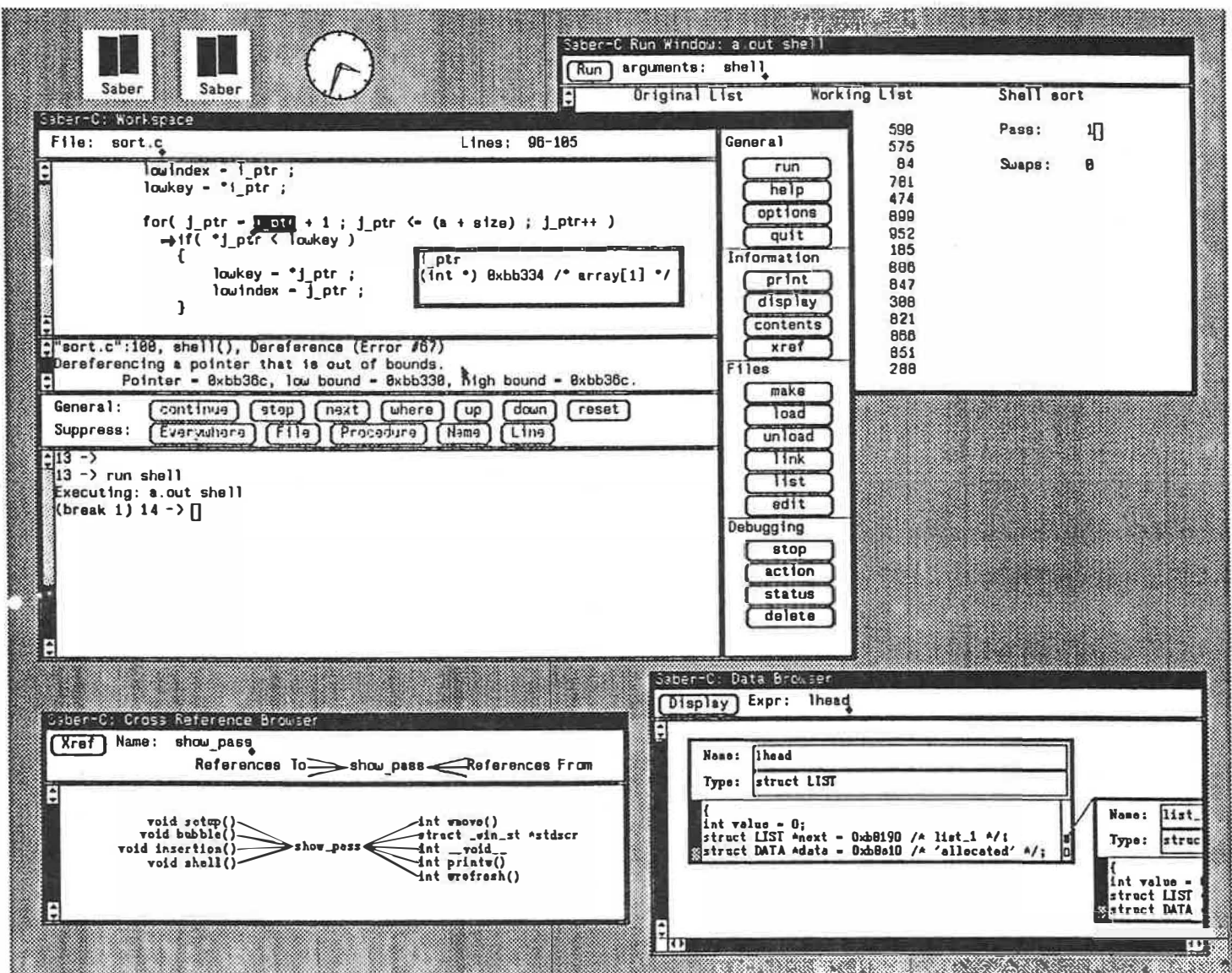
All input and output is routed through an interface manager that runs as a separate process and connects to Saber-C via a socket. The interface manager makes use of multiple windows when Saber-C is executed in a windowing environment.² While Saber-C's windowing interface is not the topic of this paper, a brief explanation is useful in order to understand the general method of interaction. Saber-C creates separate windows for program input/output, source file editing and listing, tracing, displaying user data, displaying general program information, child processes input/output, setting options, and for viewing the on-line documentation.

The screendump shown below illustrates a typical session with Saber-C. In the middle of the screen is the main Saber-C window, which consists of a source panel, a message window, and the user input window, usually referred to as the *workspace*. The panel of buttons running along the right-hand side of the window represent the most common actions taken within the Saber-C environment.

The window on the upper right of the screen is the program i/o window, where all of the input and output during program execution is sent. The window in the lower left is an invocation of the Cross Reference browser which graphically displays variable dependencies. The window in the lower right is the graphical Data browser, currently displaying the value of two structures. The remainder of this paper will focus upon the workspace window and the activities performed within it.

¹Vax is a trademark of Digital Equipment Corporation.

² The screens depicted in this paper are on the SunView windowing system. A less sophisticated multi-window display is currently available under the X windowing system while a full port of the SunView interface is being completed.



3. Sample Session

When Saber-C is started, it places the user in an interactive workspace. The workspace features a mode-less input processor that accepts both Saber-C commands and C source code. The Saber-C command set and syntax is similar to that of the UNIX debugger *dbx*. The input processor also supports a history and name completion mechanism modeled after the *tcsh* shell.

Any C statement or expression, including preprocessor directives, can be entered in the workspace. It is possible to define macros, variables, types, and functions directly in the workspace.

```
1 -> int xx ;
2 -> xx = 1 << 4 ;
(int) 16
3 -> print xx + 123
(int) 139
4 ->
```

Source code files, object code files, and library files can be loaded into Saber-C. As a file is loaded, it is dynamically linked with all previously loaded files. When a library is loaded, it is only "attached" to the workspace; the individual modules contained within the library are loaded as needed during execution.

Source files are checked for syntax violations and lint-style warnings as they are loaded. If a violation is detected, the loading process is interrupted and the location of the problem is displayed. Benign violations are reported as warnings, from which the user can continue loading the file. More severe violations are reported as errors, which require the user to correct the problem and reload the file.

```
1 -> load test.c
Loading: -Dlint -DUNIX -DSUN test.c

"test.c":4, sum(), Used before set (Warning #290)
      3:          int total ;
*      4:          total += arg_one + arg_two ;
      5:          printf("arg_one = %d, arg_two = %d, total = %d\n",
Automatic variable 'total' may be used before set.
General options: continue/silent/quit/abort/edit/reload
Suppress options: Everywhere/File/Line/Procedure/Name [c] ?
```

When a file is loaded, all symbols (variables, functions, types, and macros) defined at the global level of the file become visible in the workspace. For example, the file *test.c* defines the function *sum* and it includes the file */usr/include/stdio.h* as a header file. Once the file is loaded, it is possible to examine and use any symbols defined by *test.c* and the header files included by it.

```
2 -> sum(123, 456) ;
arg_one = 123, arg_two = 456, total = 579
(int) 579
3 ->
4 -> whatis stdin
#define stdin (&_iob[0])
5 ->
```

Saber-C automatically checks for run-time violations as a program is being executed. Possibly benign violations are reported as warnings, from which the user can continue execution. Serious violations

are reported as errors, requiring the user to correct the problem or provide a value to substitute for the incorrect expression.

```
29 -> deref(0);
About to dereference 0x0

-----
"deref.c":13, deref(), Dereference (Error #67)
   12:      printf("About to dereference 0x%lx\n", ptr);
*   13:      *ptr++ = 0;
   14:      printf("Done\n");
Dereferencing a pointer that is out of bounds.
      Pointer = 0x0, low bound = 0x0, high bound = 0x0.
Options: break/quit/edit/reload [b] ? b
(break 1) 30 ->
```

When a run-time violation occurs, execution can be suspended to create a new invocation of the workspace called a breaklevel. This breaklevel is scoped to the location where execution stopped, allowing the user to examine variables, view the execution stack, and single-step execution of the program.

```
(break 1) 30 -> where
error #67 (Dereference)
deref(ptr = (int *) 0x0) at "deref.c":13
(break 1) 31 -> ptr ;
(int *) 0x0
(break 1) 32 ->
```

While at a breaklevel, the user can execute code that may generate another breaklevel. Saber-C's support of multiple breaklevels preserves the context of each break in execution, thereby allowing the user to investigate several problems during a single run of the program.

When a load-time or run-time violation is reported, the user can choose to edit the file containing the violation. The editing interface will start the editor specified by the user's *EDITOR* environment variable. In a windowing environment, a new window is opened for each editing job. In a non-windowing environment, Saber-C mimics the behavior of *csh*, allowing the user to start, suspend, and resume edit jobs.

Files can be reloaded directly from the editor. If a violation is detected during the reloading process, the editor is automatically updated to the location of the violation. Files can be reloaded and unloaded without restriction. If a source file has been completely debugged, it can be compiled with the system compiler, and the resulting object file can be loaded in place of the source code.

Saber-C incorporates several features that control the the severity and scope of its error checking. Both load-time and run-time error detection can be suppressed interactively when a violation is reported, by embedding comments in the source code, or by using the suppress command. The reporting of violations can be suppressed individually by line, function, file, or globally. *Lint* comments (e.g. */*VARARGSn*/*) are recognized and handled appropriately.

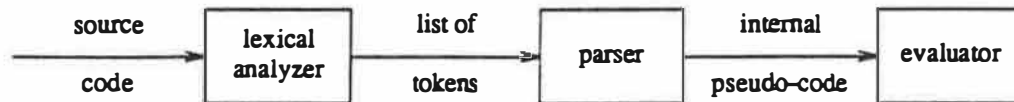
4. Interpreting C Source Code

The interpreter accepts and executes the C language as defined by K&R and as implemented by the BSD UNIX version of *pcc*. Extensions proposed by the draft ANSI standard have also been included.

4.1. The Parser

The parser uses a single pass to convert C source code into an internal pseudo-code closely resembling assembly language. This process is approximately 3 to 5 times faster than compiling the file with the UNIX C compiler.

Source code is passed through a lexer that handles all preprocessing directives and comments, and produces a list of tokens. The list of tokens is read by a LALR parser that generates pseudo-code. If the source code is from a file, the pseudo-code is saved for later evaluation. If the source code is from the workspace, it is passed directly to the evaluator.



4.2. The Evaluator

The evaluator implements a stack-based machine that executes the pseudo-code produced by the parser. The evaluator executes C code approximately 200 times slower than compiled object code; much of this speed penalty is due to the extensive error checking performed during execution.

During execution, the evaluator detects approximately 70 run-time violations involving out-of-bounds pointers, illegal array indices, memory used before set for variables and allocated data, improper function arguments, arithmetic over/underflow, and type mismatches.

In order to perform this error checking, Saber-C implements its own memory management system that is used for all statically and dynamically defined data. The memory manager maintains information about the size and type of data stored at any memory address. The size and layout of data is exactly the same as is produced by the compiler (e.g. pointers are always 4 bytes), thereby allowing data to be defined, initialized, and used by both source and object code.

The following example illustrates an out-of-bounds pointer error. When the error occurs, execution is interrupted and information about the cause and the location of the problem is displayed.

```

1 -> char *cp, buf[10];
2 -> for (cp = buf; cp <= buf + 10; cp++)
3 +>      *cp = 0;
Error #67: Dereferencing a pointer that is out of bounds.
          Pointer = 0xb7f5a, low bound = 0xb7f50, high bound = 0xb7f5a.
          Pointer went bad on line 2 in (workspace)
          Pointer previously pointed to variable buf.
  
```

The evaluator maintains information about the type of data stored at an address, enabling it to detect dynamic type mismatches. In the example below, a value of type *char ** is stored in the variable *data_instance*, only to be retrieved later as type *double*.

```

1 -> union DATA { char *name ; double value ; } ;
2 -> union DATA data_instance ;
3 ->
4 -> data_instance.name = "foobar";
5 -> printf("%f\n", data_instance.value);
Warning #54: Retrieving a <double> from data_instance.
The object stored there is a <pointer>.

```

The following example illustrates a dynamic used-before-set error. Saber-C notes in its memory manager that an address is uninitialized, allowing it to detect this violation on a byte-by-byte basis.

```

1 -> int i, *ptr ;
2 -> ptr = (int *)malloc(10 * sizeof(int)) ;
   (int *) 0xbb130 /* (allocated) */
3 -> i = *ip ;
Warning #55: Using allocated address <0xbb130> which has not been set.

```

4.3. System Environment

Since Saber-C and the user's program execute as the same process (unlike debuggers such as *sdb* and *dbx*), certain system calls are trapped to provide correct behavior during execution. For instance, all of the system calls relating to signals are handled by the interpreter so that errors in object code do not cause a core dump of Saber-C itself. Also, the functions *setjmp* and *longjmp* have been modified to allow programs to jump between object code and source code.

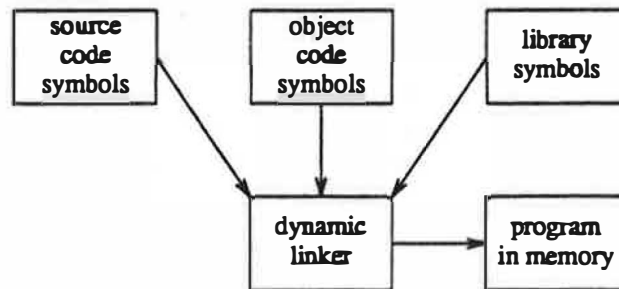
Both *fork* and *vfork* have been modified to provide better control over programs that spawn child processes. When a program calls *fork* or *vfork*, the entire Saber-C process forks. The interface manager, which executes as a separate process, also forks and reestablishes a new communication channel with the child process. In a windowing environment, Saber-C redirects the input/output streams of the evaluator to a new window. On an ascii terminal, Saber-C prompts during the forking process for a tty device to use as the console for the child process.

5. Linking Files

The dynamic linker takes the place of the UNIX linker within the Saber-C environment. As files are loaded into the environment, they are linked with all previously loaded files. The symbol and type information from source code and object code files is stored in common global tables. Saber-C allows individual files to be unloaded and reloaded without requiring that any of the other files be reloaded or relinked. Thus, the process of relinking a single file requires only a few seconds.

Libraries are initially attached to the environment, meaning that their contents are available to the linker when required. Before program execution begins, or anytime an undefined object is referenced in the workspace, the linker searches all attached libraries for definitions that will satisfy currently unresolved or undefined variables or functions.

Saber-C conforms to the behavior of the UNIX linker *ld* with regard to linking order, treatment of common symbols, and processing symbol tables and relocation information. However, while *ld* only complains when a symbol is initialized in more than one file, Saber-C also complains about size and type mismatches. When faced with a size and type mismatch, *ld* simply chooses the largest size and ignores the



type. In such similar situation, Saber-C reports a mismatch and prompts the user to either continue linking (in which case it will chose the largest size as the linker does), or abort the linking process to allow the user to correct the error.

5.1. Intermixing Source and Object code

Saber-C's linker can link together source code and object code without any restrictions. While it is relatively easy to link together data symbols, resolving *function* calls between object code and source code proved quite challenging. Functions have only one address, and function calls (including dereferencing function pointers) must work from both source and object code.

In Saber-C, object code functions remain completely untouched; however, each source code function is preceded by a header that contains several assembly code instructions. These assembly code makes a call to the 'eval' function in the evaluator, passing the name of the function to be executed. The address of a source code function is the address of the header, making it possible to pass function pointers to object code without any conversions. When a source code function calls another source code function, the header is ignored.

5.2. Prototyping

Unlike the UNIX linker, Saber-C's dynamic linker does not require that all symbols be defined before execution can begin. This greatly facilitates bottom-up and top-down development, as well as test-as-you-code debugging techniques. Saber-C maintains a list of undefined variables and functions, and propagates this information to all objects that directly or indirectly depend on the undefined entity. These dependent objects are then considered to be *unresolved*. For example, the function *test()* is unresolved if it directly calls an undefined function or uses an undefined variable, or if any execution path originating in *test()* can eventually call some undefined function or use an undefined variable.

Unresolved source code functions can be executed until the point at which an undefined variable is used or an undefined function is called. Since Saber-C cannot control execution within object code, the user is notified when entering object code that has not been fully resolved. At this point, it is possible to suspend execution and load the appropriate files or libraries to define the needed objects. Alternatively, the programmer may provide a substitute value to be used instead of calling the function or referencing the variable, thereby allowing execution to continue with the object remaining undefined.

```

1 -> int j, foo();
2 -> j = foo() ;
Error #61: Calling undefined function foo().
To continue, use 'cont <int>'.
If you define the function, use 'cont' or <Control-D> to retry.
(break 1) 3 -> cont 4
(int) 4
4 -> print j
(int) 4
5 ->

```

6. Source Language Debugging

Saber-C's source language debugger is more a collection of debugging commands than a separate component within the system. The debugging commands support breakpoints, watchpoints (suspending execution when an address is modified), tracing, and controlled execution (step and next). All of the debugging commands work at the source language level.

6.1. Extensibility

Since a fixed set of debugging commands cannot satisfy all possible debugging requests that a user could have, Saber-C's debugging commands are designed to be extensible.

All commands may be invoked through a C language interface. This involves adding the prefix "saber_" to the name of the command, and then calling it as a C function. The options and switches for the command are passed within a string as an argument to the function. The example below indicates how one might write a function to show the definition and the defining location for an identifier.

```

1 -> int show(iden)
2 +> char *iden;
3 +> {
4 +>     saber_what_is(iden);
5 +>     saber_where_is(iden);
6 +> }

```

Saber-C also provides an extensible debugging command called an *action*. The *action* works much like a breakpoint or watchpoint, but instead of stopping when the line is reached or variable modified, a user-specified section of code is executed. The text of the action can be used to test conditions, print messages, or suspend execution under certain circumstances. For example, a conditional debugging action that prints the value of two variables and suspends execution when they are equal can be specified as:

```

10 -> action in main
Setting action #1 at "main.c":50
action -> {
action ->   printf("abc = %d, xyz = %d\n", abc, xyz) ;
action ->   if( abc == xyz ) saber_stop(""); ;
action -> }
11 ->

```

In the second line of the debugging action, the C version of the *stop* command is used to suspend execution when the conditional statement is true.

A final, extensible feature of Saber-C is that users may override the default printing mechanisms for displaying data. For every unique type, the user may request that a function of their own be called whenever Saber-C would normally try and display the object. In practice, this is most commonly used to display structures according to the semantic notion of the data inside.

7. Problems and Future Directions

Saber-C is a memory and computation intensive application. Our rule of thumb is that for every megabyte of executable image, your machine should have 4 megabytes of available RAM. Saber-C will run with smaller amounts of memory, but the paging activity from the operating system quickly becomes a problem. Current development efforts are targeted at reducing Saber-C's memory requirements.

We have identified several areas in which Saber-C could be functionally improved. The debugging commands (such as *step* and *stop*) currently operate only on interpreted source code. This limitation could be eliminated by adding the capability to disassemble and debug object code.

The process of reloading a file from a breaklevel currently requires that execution start over after the reloading has completed. Ideally, users should be able to continue execution, with the newly reloaded functions replacing the old versions upon subsequent invocations.

An area that shows great potential is optimizing the intermediate code produced by the parser. The current intermediate code is based on a complex instruction set that requires the evaluator to perform several lookups for even simple C statements. Simplifying the intermediate code could improve the speed of execution significantly.

8. Conclusion

At the time this paper is being written, Saber-C is being used by several hundred users. Saber-C has been used to help develop software such as the X11 Toolkit at MIT's Project Athena, the Diamond Multimedia Editor at Bolt Beranek & Newman in Cambridge, as well as, of course, development of Saber-C itself.

Saber-C has proven to be a practical tool for developing software in the C language. The interactive workspace provides a unique run-time environment for testing and debugging C code. The ability to experiment with arbitrary C expressions or subsections of a program is extremely useful for prototyping and modular development of programs.

Saber-C's comprehensive load-time and run-time error detection reduces the difficulty of locating program errors. The debugging commands provide users with complete source language debugging. Also, the editing interface facilitates easy movement between the workspace and an edit job.

References

- [1] Adams, E. and Muchnick, S.S., "Dbxtool: A window-based symbolic debugger for Sun workstations," *Software - Practice and Experience*, vol. 16, no. 7, July 1986, pp. 653-669.
- [2] Chase, B. B. and Hood, R. T., "Selective interpretation as a technique for debugging computationally intensive programs", *ACM SIGPLAN Notices*, vol. 22, no. 7, July 1987, pp. 113-124.
- [3] DeLisle, N.M. et Al., "Viewing a programming environment as a single tool," *ACM SIGPLAN Notices*, vol. 19, no. 5, April 1984, pp. 49-56.
- [4] Feuer, A. R., "si - an interpreter for the C language," *USENIX Conference Proceedings*, Summer 1985, pp. 47-55.
- [5] Feuer, A. R., "Introduction to the Safe C Runtime Analyzer," *Catalytix Corporation Technical Report*, January 1985.
- [6] Ross, G., "Integral C - a practical environment for C programming," *ACM SIGPLAN Notices*, vol. 22, no. 1, January 1987, pp. 42-48.

Associative arrays in C++

Andrew Koenig

AT&T Bell Laboratories

184 Liberty Corner Road; Warren NJ 07060; ark@europa.att.com

ABSTRACT

An *associative array* is a one-dimensional array of unbounded size whose subscripts can be non-integral types such as character strings. Traditionally associated with dynamically typed languages such as AWK and SNOBOL4, associative arrays have applications ranging from compiler symbol tables to databases and commercial data processing.

This paper describes a family of C++ class definitions for associative arrays, including programming examples, application suggestions, and notes on performance. It concludes with a complete programming example which serves the same purpose as a well-known system command but is much smaller and simpler.

1. Introduction

An *associative array* is a data structure that acts like a one-dimensional array of unbounded size whose subscripts are not restricted to integral types: it associates values of one type with values of another type (the two types may be the same). Such data structures are common in AWK^[1] and SNOBOL4^[2], but unusual in languages such as C^[3], which have traditionally concentrated on lower-level data structures.

Suppose that *m* is an associative array, or *map*, whose subscripts, or *keys*, are of type `String` and whose elements, or *values*, are integers. Suppose further that *s* and *t* are `String` variables. Then *m[s]* is an `int` called *the value associated with s*. In this example *m[s]* is an *lvalue*: it may be used on either side of an assignment. If *s* and *t* are equal, *m[s]* and *m[t]* are the same `int` object. If *s* and *t* are unequal, *m[s]* and *m[t]* are different objects.

This is exactly analogous to the way ordinary arrays work in most languages, except that it is much harder to pre-allocate enough memory in advance to contain all relevant elements. Thus some kind of dynamic memory scheme is necessary to allow a map to accommodate all its elements.

C data structures do not expand automatically, and no such expanding structures are built into C++ either. But C++^[4] is extensible: although it does little to provide high-level data structures by itself, it makes it possible to implement such structures in ways that are easy to use.

1.1 An example

The following C++ program uses associative arrays to count how many times each distinct word occurs in its input:

```

#include <String.h>
#include <Map.h>

Mapdeclare(String,int)
Mapimplement(String,int)

main()
{
    Map(String,int) count;
    String word;

    while (cin >> word)
        count[word]++;

    Mapiter(String,int) p (count);
    while (++p)
        cout << dec(p.value(),4) << " " << p.key() << "\n";
}

```

The `#include` statements bring in declarations for a locally-written string package^[5] and the author's associative array package, respectively. The string package requests `<stream.h>` so we do not have to name it explicitly.

`Mapdeclare(String,int)` defines two new types: `Map(String,int)` to represent an array of integers whose keys are of type `String` and `Mapiter(String,int)` to represent an element of such an array.

The first declaration in the main program thus defines `count` to be of type `Map(String,int)`.

The program does most of its work in the first loop. It reads each word from its input into `word` and increments the corresponding element of `count`. Trying to increment a non-existent element automatically creates the element first and gives it an initial value of 0. This value is almost immediately incremented to 1.

When the input is exhausted, the first loop finishes. All the information we need is now in `count`; we need merely print it. To do this, we use a *map iterator*.

A map iterator refers to a particular map (associative array) and usually to a particular element of that map. The declaration

```
Mapiter(String,int) p (count);
```

defines `p` to be an iterator that will refer to an element of `count`. Since we did not specify a particular element, `p` is initially *vacuous*: it does not refer to any element.

Incrementing a vacuous map iterator causes it to refer to the first element of the map; testing it in a `while` statement yields *true* as long as the iterator is not vacuous. Thus the final loop steps `p` through all the elements of `count`.

Every map element has two parts: a *key* and a *value*. In the final statement we print `p.value()` and `p.key()`, using the `dec` function to get the output to line up neatly.

Here is the result, collected into two columns, of giving a copy of this program as input to itself:

```

2 "                                1 "\n";
2 #include                        1 (++p)
1 (cin                            1 (count);
4 <<                              1 <Map.h>
1 <String.h>                      1 >>
1 Map(String,int)                 1 Mapdeclare(String,int)
1 Mapimplement(String,int)        1 Mapiter(String,int)
1 String                          1 count;
1 count[word]++;                  1 cout
1 dec(p.value(),4)                1 main()
1 p                               1 p.key()
2 while                           1 word)
1 word;                           1 {
1 }

```

The program is obviously naive about the role of punctuation, but the output is otherwise about what one would expect. Notice especially that the output is sorted by key. The ordering, of course, depends on the particular C++ implementation's collating sequence for character strings. In ASCII, special characters come before letters and upper-case letters come before lower-case letters. Thus `while` appears before `word` and `String` comes before `count`.

2. Defining Map classes

Every file in every program that uses a map class must include `<Map.h>`. This header file defines four macros: `Mapdeclare`, `Mapimplement`, `Map`, and `Mapiter`. Each of these macros takes two type names as arguments representing the key and value types for a particular map class. Thus we use `Map(int,double)` as the class name of a map with `int` keys and `double` values, `Map(String,String)` as the class name of a map with `String` keys and `String` values, and so on.

2.1 Key and value types

There are several restrictions on the key and value types:

- Each type name must be expressed as a single identifier. Thus one cannot refer to `Map(char*,int)`: `char*` is not an identifier. Use a `typedef` declaration to circumvent this restriction.
- Each type must have *value semantics*. A map stores copies of the values given it, so it must be possible to copy such values. Moreover, a value stored in a map must persist, and be valid, as long as the map itself exists. Thus even if `Map(char*,int)` were syntactically possible, it would be a bad idea unless extreme care were taken to preserve the memory referenced by each pointer stored in the map. The `String` class does have value semantics and is therefore ideal for use in maps. Other types with value semantics include `int`, `char`, `long`, and `double`.

There is no way for the map package to check the semantics of its underlying types. Hence trying to define a map on types of inappropriate semantics usually leads to disaster.

- Neither the key nor the value type may require arguments in its constructor, although either type may permit them. Equivalently, it must be possible to declare an object of each type without initializing it.
- The key type should not contain any underscores in its name. Failure to observe this restriction might cause two types that are conceptually different to generate the same internal name. This is consequence of implementation details that are beyond the scope of this paper.

In addition to these restrictions, it must be possible to compare keys for sorting purposes. In other words, if `k1` and `k2` are two keys, `k1<k2` must be well-defined. Moreover, when `<` is applied to two keys, it must be a *strong total order relation*. That is, it must have the following properties:

- $k_1 < k_1$ must never hold for any key k_1 .
- For any two keys k_1 and k_2 , either $k_1 < k_2$, $k_2 < k_1$, or k_1 and k_2 are equal. No more than one of these conditions may hold for any particular values of k_1 and k_2 .
- For any three keys k_1 , k_2 , and k_3 , if $k_1 < k_2$ and $k_2 < k_3$ both hold then $k_1 < k_3$ must also hold.

Notice that we do not require that $k_1 == k_2$ or $k_1 > k_2$ be defined because we can infer $k_1 > k_2$ from $k_2 < k_1$ and $k_1 == k_2$ from $!(k_1 < k_2) \wedge k_2 < k_1$.

The integral types (int, short, char, and long) and the String class all have an appropriate definition for $<$. The floating-point types are a different matter: some machines support a special floating-point “not-a-number” value (NaN) that is neither less than, greater than, or equal to anything else. Using floating-point keys on such a machine invites disaster if a key is ever NaN. Floating-point *values* cause no trouble.

2.2 Mapdeclare and Mapimplement

If K is the key type and V is the value type, saying

```
Mapdeclare(K,V)
```

declares the classes $\text{Map}(K,V)$ and $\text{Mapiter}(K,V)$. Thus the user must use `Mapdeclare` in every file in the program that uses either of these classes.

However, `Mapdeclare` includes only the declarations of these classes. It does not include the *definitions* of the classes' member functions: those appear in `Mapimplement`. Thus every program that uses $\text{Map}(K,V)$ or $\text{Mapiter}(K,V)$ *anywhere* must use `Mapimplement(K,V)` in *exactly one file* of the program. This file must also use `Mapdeclare(K,V)`.

Of course, if K or V is itself a class, the class definition must appear before the use of `Mapdeclare`.

For instance, we might split our sample program into two files. One file uses `Map(String,int)` and must therefore declare it:

```
#include <String.h>
#include <Map.h>

Mapdeclare(String,int)

main()
{
    Map(String,int) count;
    String word;

    while (cin >> word)
        count[word]++;

    Mapiter(String,int) p (count);
    while (++p)
        cout << dec(p.value(),4) << " " << p.key() << "\n";
}
```

The other file defines the map class member functions:

```
#include <String.h>
#include <Map.h>

Mapdeclare(String,int)
Mapimplement(String,int)
```

These two files can be separately compiled and linked together with the same effect as the original

sample program. Doing this is only of pedagogical interest for a program this small, but is important for larger programs.

In subsequent code fragments we will omit `#include` statements and uses of `Mapdeclare` and `Mapimplement`.

2.3 Using a map

Indexing is the fundamental map operation: if `m` is of type `Map(K,V)` and `x` is of type `K`, then `m[x]` is an lvalue of type `V`. Thus we store a value `v` in `m[x]` by writing

```
m[x] = v;
```

and retrieve it again by writing

```
v = m[x];
```

If we ask for the value of `m[x]` without ever having specified one, we get the *default value* of type `V`: the value of an otherwise uninitialized static variable of type `V`. It is very important that this is the value of a *static* variable in case `V` is a built-in type: the default value of an ordinary `int` variable is undefined but static `int` variables are initialized to zero. Thus after executing:

```
Map(String,int) m;
```

```
m["Morristown"]++;
```

`m` has one element with key `Morristown` and value 1.

Because evaluating `m[k]` creates an element of `m` if an appropriate element does not already exist, we need a way to test whether an element exists without creating one. Thus every map has a member function called `element` which returns an appropriate value:

```
if (m.element(k))
    // m[k] exists
```

Thus if we did not wish to rely on default values for map elements, we could write:

```
if (m.element(word))
    m[word]++;
else
    m[word] = 1;
```

The number of elements in a map `m` is `m.size()`.

We can explicitly nominate a default value for map elements by passing that value to the map constructor. For example:

```
Map(String,String) s ("(undefined)");
```

declares `s` to be a map in which keys and values are both of type `String` and in which element values are initially `(undefined)`.

2.4 Copying a map

If `m` is a map,

```
Map(K,V) n = m;
```

will create a new map `n` whose elements are logical copies of the elements of `m` (that is, the result of applying operator=). The default value for subsequently created elements of `n` will be the same as for `m`. The same effect can be obtained by writing:

```
Map(K,V) n (m);
```

If `m` and `n` are maps with the same key and value types, then

```
m = n;
```

effectively discards all the elements of `m` and creates new elements that are logical copies of all the elements of `n`. This does not change the default value for subsequently created elements of `m`; the value, if any, given when `m` was created remains.

One consequence of this is that passing a map as a function argument copies all the elements of the map. This preserves the normal call-by-value semantics of C++ but may have unexpected performance consequences. Consider writing functions whose arguments are references to maps instead of maps themselves.

3. Iterators

Every map class `Map(K,V)` has a companion *map iterator* class `Mapiter(K,V)` which is useful for extracting elements from a map. A map iterator refers to a particular map and optionally to a particular element of that map. For example:

```
Map(String,int) m;
Mapiter(String,int) mi (m);
```

These declarations create a map `m` and an iterator `mi` that can refer to an element of `m`. We did not specify any element in this example, so `mi` doesn't refer to one right now. Thus we call `mi` *vacuous*.

We can make `mi` identify the element of `m` with key `k` by saying

```
mi = m.element(k);
```

If `m[k]` exists, `mi` now identifies it; otherwise `mi` is vacuous.

Thus we see that `element` actually yields an iterator. We can tell whether an iterator is vacuous by using it as the subject of a test:

```
if (mi)
    // mi is not vacuous
```

Assigning a map to an iterator yields a vacuous iterator. Thus, after executing:

```
mi = m;
```

`mi` is a vacuous iterator referring to `m`.

Map elements are always kept sorted by key. Thus it makes sense to ask about the lowest and highest keys. For a map `m`, `m.first()` is an iterator that refers to the first element (the element with the lowest key) and `m.last()` is an iterator that refers to the last element (the element with the highest key). If `m` is empty, both `m.first()` and `m.last()` are vacuous.

If `mi` refers to an element, `++mi` makes `mi` refer to the element whose key immediately follows the current one. Similarly `--mi` makes `mi` refer to the immediately preceding key.

If `mi` is vacuous, `++mi` effectively assigns `m.first()` to `mi` and `--mi` effectively assigns `m.last()` to `mi`. If `mi` refers to `m.last()`, `++mi` makes `mi` vacuous, and if `mi` refers to `m.first()`, `--mi` makes `mi` vacuous. Thus we can think of all the keys in a map as being circularly connected, with a vacuous "key" marking a starting (and ending) point in the circle.

At this point we know at least two ways to refer to every element in a map:

```
for (Mapiter(K,V) mi = m.first(); mi; ++mi)
    // do something with mi
```

or

```
for (Mapiter(K,V) mi = m; ++mi; )
    // do something with mi
```

but we need a way to get at the key and value parts of the element referenced by `mi`. To this end,

map iterators have member functions called `key` and `value`. Thus we might write:

```
for (Mapiter(K,V) mi = m.first(); mi; ++mi) {
    K k = mi.key();
    V v = mi.value();
    // do something
}
```

If `mi` is vacuous, `mi.key()` and `mi.value()` are copies of static variables of type `K` and `V` respectively, to forestall core dumps and other disasters.

4. Performance

Associative arrays greatly simplify many programming tasks, but that is almost irrelevant if they are too expensive to use. Thus, while a map will never be as fast as an ordinary array, we have gone to some effort to ensure that the map classes perform reasonably well. In general, for a map with n elements,

- Searching for an element takes $O(\log n)$ time.
- Inserting a new element takes $O(\log n)$ time plus the time to call `malloc` once.¹
- Incrementing or decrementing an iterator takes $O(\log n)$ time. Visiting every element of a map takes $O(n)$ time (not $O(n \log n)$).
- Copying an entire map takes $O(n \log n)$ time.

These are worst-case times: there is no pathological input that might cause the performance to degrade beyond these bounds.

A map takes a trivial amount of space for the map itself plus an additional overhead per element of three pointers, one character, and whatever additional overhead `malloc` imposes. Note that maps only require memory to be allocated for the elements actually stored.

What about absolute execution time?

As an example, we wrote a small program to generate 20,000 random integers and store them in a conventional array and a map. On our system (A DEC MicroVAX II running the Ninth Edition of the UNIX system), it takes 0.9 seconds to fill the conventional array and 8.8 seconds to fill the map, including the time spent in the `rand` function. So in this application maps are about ten times slower than conventional arrays.

But of course maps are not intended to replace conventional arrays where speed is all that matters — they do other things as well. For instance, storing values in a map has the side effect of sorting them by key. Thus using the random numbers as keys sorts them for free. Picking them out of the map again takes 0.8 seconds. In other words, using a map to sort 20,000 random integers takes 9.6 seconds: 8.8 seconds to put them in and 0.8 to take them out again.

In contrast, using the `qsort` routine to sort the conventional array takes 17 seconds.

5. Suggested applications

Although a map is not a good replacement for a conventional array when time is critical, time is often not critical. For example, we have seen many programs that use conventional arrays to build

1. On some systems, `malloc` runs in time proportional to the number of blocks already allocated. We believe this could cause performance problems for some applications on such systems. The next version of C++ provides a clean way to define special-purpose `new` and `delete` operators for a particular class; we will consider that facility when it is widespread.

up an array of arguments to hand to some system command or other utility routine. The usual approach is to make the array some arbitrary size and then either not check for overflow (so that the program aborts unpredictably when overflow occurs) or check for it and terminate the program explicitly.

A map can often be used in this context as if it were a conventional array of unbounded size. Even if the utility being called demands a conventional array as its input, it is possible to use the `size` function to allocate the right amount of memory at the last minute and then copy the elements. This approach may waste some machine resources, but the waste is often negligible compared to the execution time of the utility itself and carries a compensating gain in generality and robustness.

Maps are a good candidate for use in compiler symbol tables. In particular, it is unnecessary to sort such a symbol table in order to print a cross-reference listing.

The value of a map element can itself be a map (use `typedef` to get around the single-identifier restriction). If `m` is such a map, `m[k1][k2]` has the expected meaning: `m[k1]` is the map that is the value of the element of `m` with key `k1`, and `m[k1][k2]` is the value of the element of `m[k1]` with key `k2`.

6. Implementation notes and warnings

These classes are presently implemented as AVL trees. An AVL tree is a binary tree with the added requirement that the heights of the two subtrees of each node may differ by no more than one^[6]. Although an algorithm exists for efficient deletion from an AVL tree, it is quite a nuisance and we have not implemented it in the current version.

We believe our implementation is reasonably robust. There are, however, a few places where a user might get into trouble:

- If `m` is a map, referring to `m[i]` creates it if it did not already exist. It is therefore a bad idea, though a tempting one, to check whether a map element exists by comparing its value to the default value. Use the `element` function instead.
- A map iterator is conceptually a pointer. Like any pointer, an iterator invites the possibility of a dangling reference if the lifetime of an iterator exceeds that of the associated map. For example:

```
Map(K,V)* p;
p = new Map(K,V);
Mapiter(K,V) mi (*p);
delete p;
```

Because it is impossible to create an iterator without making it refer to some map, it is somewhat tricky to step into this particular trap. Nevertheless it is possible.

- Copying a map makes a logical copy of each element. If the map is large, this can take a long time. Many functions that deal with maps can deal just as profitably with pointers or references to maps instead.
- Data types used in maps should only be pointers if it is possible to ensure that those pointers refer to memory at least as persistent as the maps themselves. Otherwise some map element will ultimately contain a pointer to garbage, with chaos as the result.

7. Another example: topological sorting

We conclude with a non-trivial programming example that uses maps, strings, and lists. We will say enough about strings and lists in passing to enable the reader to follow the exposition; construction of appropriate class definitions is an interesting exercise for the reader.

7.1 The problem

Our input file consists of words separated by white space (spaces, tabs, newlines). Each word is considered to represent some abstract object; the words are taken in pairs to assert that the first object of each pair is "less than" the second. If both words in the pair are the same, that pair just says that the particular word exists without saying anything about its ordering. The problem is to find a single linear ordering that satisfies all the given constraints. This may be impossible because of a loop in the ordering. In that case, the program should say so.

For example, the following input represents some constraints on clothing:

```
pants      belt
left_sock  left_shoe
right_sock right_shoe
shirt      shirt
pants      left_shoe
pants      right_shoe
```

Here, we say that one article of clothing is "less than" another if it must be put on before the other. Thus each sock must be put on before the corresponding shoe, it is difficult to put on pants over shoes, one must put one's pants on before one's belt, and there is a shirt in there somewhere.²

One ordering that satisfies all these constraints is:

```
left_sock
pants
right_sock
shirt
belt
left_shoe
right_shoe
```

There are others, of course.

Topological sorting is often used to put modules into an appropriate order for load module archives in the UNIX® system; there is therefore a topological sort utility program called `tsort` available as a standard for comparison.

7.2 Implementation

We use an algorithm that is easy to understand and describe.^[7] we read pairs of items, or *tokens*, keeping for each token a *count* of its predecessors and a *list* of its successors. Tokens with no predecessors can be written out immediately. When we write a token, we decrement the predecessor count of all its successors; decrementing a count to zero means that token can then be written. If we run out of tokens to write before we've dealt with every token, that means there is a loop somewhere.

Here are the details. First we bring in header files and say that we are going to want to use lists of strings:

2. The author does not ordinarily wear a tie.

```

#include <String.h>
#include <list.h>
#include <Map.h>

listdeclare(String)
listimplement(String)

```

The list package works similarly to the map package: the `listdeclare` and `listimplement` macros say that we are going to want to use lists of `String` objects.

Now we define a structure to represent the information we need to store about an object: a count of its predecessors and a list of its successors:

```

struct token {
    int predcnt;
    String_list succ;
};

```

A `String_list` is a list of `String` objects, as defined by the `listdeclare` macro.³

Having defined a structure with the information we need, we can now define a map whose keys are strings and whose values are tokens:

```

Mapdeclare(String,token)
Mapimplement(String,token)

```

We are ready to do some real work. The program first reads pairs of tokens. If both tokens in the pair are identical, we just record that the token exists. Otherwise, we increment the predecessor count of the second one and add the second to the successor list of the first. Here we use the `put` member function of a list, which appends its argument to the end of the list.

```

Map(String,token) m;
String p, s;

while (cin >> p >> s) {
    if (p == s)
        (void) m[p];
    else {
        m[s].predcnt++;
        m[p].succ.put(s);
    }
}

```

We look at each token and make zeroes into a list of the names of the tokens with no predecessors:

3. As written, this example requires a version of the C++ translator that is still experimental; that version defines assignment and initialization of structures in terms of assignment and initialization of their elements, even if those elements have constructors.^[6] For present versions of C++, we write it as follows:

```

struct token {
    int predcnt;
    String_list succ;
    void operator= (token& t) { predcnt = t.predcnt; succ = t.succ; }
    token (token& t): predcnt (t.predcnt), succ (t.succ) {}
    token() { }
};

```

```
String_list zeroes;

for (Mapiter(String,token) i = m.first(); i; ++i) {
    if (i.value().predcnt == 0)
        zeroes.put(i.key());
}
```

Next we print the tokens on the list. Each time we print a token, we decrement the predecessor count of its successors. If a count reaches zero, we add that token to the list. Along the way, we count how many tokens we print.

To do this, we use the `getX` function, which removes an element from the beginning of a list, places the element in its argument (which is a `String&` in this case), and returns zero (false) or nonzero (true) to reflect the success of the operation.

```
int n = 0;

while (zeroes.getX (p)) {
    cout << p << "\n";
    n++;
    String_list& t = m[p].succ;
    while (t.getX (s)) {
        if (--m[s].predcnt == 0)
            zeroes.put (s);
    }
}
```

Finally, we check whether we have printed every token:

```
if (n != m.size())
    cout << "the ordering contains a loop\n";
```

7.3 Performance

This straightforward program is about a quarter the size (in lines of source code) of `tsort`, so one should not be surprised to find that it runs substantially more slowly.

In practice, however, this is not the case. For very small inputs, `tsort` is faster because `String` input is quite slow in our present C++ implementation.⁴ However, for larger inputs, the logarithmic behavior of maps wins over the quadratic behavior of the algorithm used in `tsort`.⁵

As an example, we used two sets of test data. Because topological sorting is often used on program dependencies, we generated our test data from two actual load module libraries: the C library on our system and the PORT library of mathematical software. Here are the results:

4. This problem is understood; current work aims to reduce the overhead.

5. To forestall the objection that it is unfair to compare a quadratic algorithm to one that is intrinsically faster, we note that (a) the present implementation of `tsort` was considered "not worth improving" for years, and (b) when David Gay finally improved it on the Ninth Edition system, his (very fast) improvement was about ten times the size of our example.

	C	PORT
total relations	742	6,513
distinct tokens	248	1,505
tsort execution time	7.0	406.5
C++ program execution time	9.3	101.8
input overhead	4.9	39.5

The first two lines in the table characterize the size of the problem in terms of the total number of relations and the number of distinct tokens in the input data. The next two lines give the execution time, in seconds, to process each set of data. Note that the C++ program is slightly slower on the smaller sample and much faster on the larger sample.

The last line shows how much of the execution time of the C++ program can be attributed to input overhead. This shows that if we could eliminate input overhead entirely, we could more than double the speed of the C++ program (remove 4.9 out of 9.3 seconds) on the small sample and significantly reduce it (remove 39.5 out of 101.8 seconds) on the large one. There are several ways to do this (aside from the obvious one of waiting for the library to improve); it is left as an exercise for the reader.

8. Conclusion

We have described a C++ implementation of a high-level data structure. The availability of this structure led us to demonstrate it by a simple solution to a well-known problem. Our solution turned out to be a quarter the size of the "standard" one. Moreover, its performance on large input data was substantially better as well.

REFERENCES

1. Aho, Kernighan, and Weinberger: *The AWK Programming Language*; Addison-Wesley, 1988.
2. Griswold, Poage, and Polonsky: *The SNOBOL4 Programming Language*; Prentice-Hall, 1971.
3. Kernighan and Ritchie: *The C Programming Language*; Prentice-Hall, 1978
4. Stroustrup: *The C++ Programming Language*; Addison-Wesley, 1986
5. J. E. Shopiro: *Strings and Lists for C++*, unpublished internal memorandum
6. D. E. Knuth: *The Art of Computer Programming*, volume 3, section 6.2.3, page 451.
7. D. E. Knuth: *The Art of Computer Programming*; Volume 1, page 262
8. Stroustrup, *The Evolution of C++: 1985 to 1987* proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico, 1987; pp. 1-21.

NAME

Map — associative array classes

SYNOPSIS

```
#include <Map.h>

Mapdeclare(S,T)
Mapimplement(S,T)

class Map(S,T);
class Mapiter(S,T);
```

DESCRIPTION

Every *map* is a collection of *elements*, each of which contains a *key* part of type *S* and a *value* part of type *T*. In order for these values to be usefully stored in an element, both *S* and *T* must have *value semantics*: it must be possible to assign or initialize an object of type *S* or *T* with the effect of copying it. Because assigning a pointer does not copy the associated memory, *S* and *T* should not be pointer types unless the only significance of the pointers is the pointer values themselves and not the memory they address. In particular, *S* and *T* *should almost never be character pointers*.

The elements of a map all have distinct keys. Thus it must be possible to compare values of type *S*. Moreover, map elements are ordered by key, so a strong total order relation $<$ must be defined on values of type *S*. It is unnecessary for $==$ or any of the other usual relations to be defined. *T* need not have an order relation.

The key and value types should ideally be *type parameters*; the C preprocessor presently simulates this facility. Thus the type name of a map takes the form `Map(S,T)`, where *S* and *T* are type names that are single identifiers. *S* is called the *key* type and *T* is the *value* type. For example, `Map(String,int)` represents an associative array with keys of type `String` that contains `int` values.

The macro `Mapdeclare(S,T)` generates the declarations for classes `Map(S,T)` and `Mapiter(S,T)` (described later). It must appear once in every source file for every combination of *S* and *T* used in that source file. The macro `Mapimplement(S,T)` generates the program text that implements the map classes. It must appear exactly once in the entire program for each combination of *S* and *T* used in the program. Because `Map(S,T)` and `Mapiter(S,T)` are preprocessor macros, no spaces may appear in these names.

If *i* is of type *S*, *x* of type *T*, and *m* and *n* of type `Map(S,T)` with *p* and *q* elements, respectively, the following constructors (and related declarators) are defined:

`Map(S,T) ()` An empty map. The value part of future elements is the value of an otherwise uninitialized static object of type *T*.

`Map(S,T) (x)` An empty map whose future elements have default value *x*.

`Map(S,T) (m)` A copy of *m* obtained by copying the elements and default value of *m*.

The following operators and functions are defined:

`n = m` All the elements of map *n* are deleted and new elements are added to *n* that are copies of the elements of *m*. The default value part of future elements of *n* does not change. The result, of type `Map(S,T)&`, is the new value of *n*. Runs in $O(\log(p) + \log(q))$.

`m[i]` A reference, of type `T&`, to the value part of the element of *m* with key *i*. If the element does not exist, it is created. Runs in $O(\log(p))$.

`m.size()`

 An `int` containing the number of elements in *m*. Runs in $O(1)$.

`m.element(i)`

 A map iterator (of type `Mapiter(S,T)`, described later) referring to the element of *m*

with key *i* if such an element exists. If no such element exists, the result is *vacuous*. Runs in $O(\log(p))$.

m.first()

A map iterator (of type `Mapiter(S,T)`) referring to the element of *m* with the lowest key. If *m* has no elements, the result is *vacuous*. Runs in $O(\log(p))$.

m.last()

A map iterator (of type `Mapiter(S,T)`) referring to the element of *m* with the highest key. If *m* has no elements, the result is *vacuous*. Runs in $O(\log(p))$.

Map iterators

For every class `Map(S,T)` there is a class `Mapiter(S,T)` that can be used to identify an element of a `Map(S,T)` object. Objects of class `Mapiter(S,T)` are called *map iterators*. Every map iterator identifies a particular map and, optionally, a particular element of that map. A map iterator that does not identify an element is *vacuous*. If a map iterator is used as a test subject for an *if*, *while*, or *for* statement, it is considered false if it is *vacuous* and true otherwise.

If *m* is of type `Map(S,T)` with *p* elements and *i* and *j* are of type `Mapiter(S,T)`, the following constructors (and associated declarators) are defined:

Mapiter(S,T) (m)

Create a *vacuous* iterator referring to *m*. Runs in $O(1)$.

Mapiter(S,T) (i)

Create an iterator referring to the same map and element as *i*. Runs in $O(1)$.

The following operators and functions are defined:

j = i Make *j* refer to the same map and element as *i*. The result, of type `Mapiter(S,T)&`, is the new value of *j*. Runs in $O(1)$.

i.key()

A value of type *S* that is the key part of the element referred to by *i*. If *i* is *vacuous*, *i.key()* is the value of an otherwise uninitialized static object of type *S*. Runs in $O(1)$.

i.value()

A value of type *T* that is the value part of the element referred to by *i*. If *i* is *vacuous*, *i.value()* is the value of an otherwise uninitialized static object of type *T*. Runs in $O(1)$.

++i If *i* is *vacuous*, make it refer to the map element with the lowest key. Otherwise, make it refer to the map element with the lowest key greater than the element to which *i* presently refers. If no appropriate element exists, *i* becomes *vacuous*. The result, of type `Mapiter(S,T)&`, is the new value of *i*. Runs in $O(\log(p))$.

--i If *i* is *vacuous*, make it refer to the map element with the highest key. Otherwise, make it refer to the map element with the highest key less than the element to which *i* presently refers. If no appropriate element exists, *i* becomes *vacuous*. The result, of type `Mapiter(S,T)&`, is the new value of *i*. Runs in $O(\log(p))$.

Although a single *++i* or *--i* run in $O(\log(p))$ time, using an iterator to access all elements of *m* takes only $O(p)$, not $O(p \log(p))$.

BUGS

There is no way to delete an element. It is the user's responsibility to stop using a map iterator after the corresponding map has disappeared. Ambiguities can occur if the type name *S* contains an underscore. A core dump is likely if memory is exhausted unless someone has provided an error handler for operator *new*.

Copy-on-Write For Sprite

Michael Nelson
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

The Sprite operating system uses copy-on-write during process creation. The mechanism is a combination of copy-on-write (COW) and copy-on-reference (COR), which simplifies the implementation and reduces the cache flushing overhead on machines with virtually-addressed caches. The COW-COR mechanism can potentially improve fork performance over copy-on-fork schemes from 10 to 100 times depending on whether pages are resident in memory or on backing store. However, in normal use more than 70% of the pages have to be copied anyway. With the Sprite implementation the overhead of handling the additional page faults required results in worse overall performance than copy-on-fork; with a more optimized implementation forks would be about 20% faster with COW-COR than with copy-on-fork. A pure copy-on-write scheme would eliminate 10 to 20 percent of the page copies required under COW-COR and would provide up to a 20% improvement in fork performance over COW-COR. However, because of extra cache-flushing overhead on machines with virtually-addressed caches, copy-on-write may have worse overall performance than COW-COR on these types of machines.

1. Introduction

In systems that create new processes by forking, one of the major costs of process creation is copying the address space from the parent to the newly created child. A common method of improving the performance of process creation is by using copy-on-write: pages in the address space are initially shared by the parent and child; a page isn't actually copied until one of the processes attempts to modify it. Copy-on-write saves not only copying of pages in memory, but also copying of pages that are on backing store. Copy-on-write has been implemented in several systems, with the earliest being TENEX [BOBR72,MURP72] and one of the most recent being Mach [RASH87].

This paper describes a simple copy-on-write mechanism that has been implemented as part of the Sprite operating system [OUST88]. It differs from other copy-on-write mechanisms in that it is actually a combination of copy-on-write (COW) and copy-on-reference (COR); for each page that is involved in copy-on-write activity, one segment has it copy-on-write and all other segments that reference it have it copy-on-reference. We chose the COW-COR mechanism for two reasons: virtually-addressed caches and simplicity. The SPUR hardware [HILL86], which is one of Sprite's target machines, uses virtually-addressed caches that do not provide efficient support

This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

for copy-on-write; expensive cache flushing operations are required in order to implement copy-on-write on SPUR. The Sprite COW-COR scheme can be implemented on architectures such as SPUR with less cache flushing overhead than a pure copy-on-write scheme.

The other major reason for using the Sprite scheme was simplicity. One of the major complexities of copy-on-write is handling the tree of descendants that results from a single parent. In the Sprite scheme, this potentially complex tree structure is represented by a simple linear list. This simplification and others made the addition of copy-on-write to Sprite an easy task; the implementation was completed in less than one man-week.

In order to compare the Sprite COW-COR scheme to copy-on-fork schemes, we measured the performance of the Sprite COW-COR scheme by running benchmark programs against Sprite and by monitoring normal use of the system. The measurements indicate that the COW-COR mechanism can potentially improve fork performance over copy-on-fork schemes from 10 to 100 times depending on whether pages are resident in memory or on backing store. However, during normal use, the COW-COR mechanism provides only a small benefit: less than 30 percent of page copies are eliminated. With the Sprite implementation the overhead of handling the additional page faults required results in worse overall performance than copy-on-fork; a more optimized implementation could provide more than a 20% improvement in performance over copy-on-fork. A pure copy-on-write scheme would eliminate 10 to 20 percent of the page copies required under COW-COR and would provide up to a 20% improvement in fork performance over COW-COR. However, because of extra cache-flushing overhead on machines with virtually-addressed caches, copy-on-write may have worse overall performance than COW-COR on these types of machines.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the Sprite virtual memory system; Section 3 discusses previous work and Mach in particular; Section 4 describes the Sprite copy-on-write mechanism; Section 5 compares the Sprite scheme to a pure copy-on-write scheme; and Section 6 gives measurements of the performance of the Sprite scheme.

2. Sprite Virtual Memory

A Sprite process's virtual address space is divided up into three segments: code, heap and stack. Each segment has its own file that is used for backing store and its own page table that describes the segment's virtual address space. Segments can be shared by different processes. When a process forks, the child will share the parent's code segment read-only, the child is given a virtual copy of the stack segment, and the heap segment is either write-shared or a virtual copy is given to the child. A copy-on-write mechanism has the potential of saving the physical copying of pages in the stack and heap segments.

3. Previous Work

The original idea of copy-on-write emerged over 15 years ago with TENEX [BOBR72, MURP72]. Since then it has been implemented in several systems [AKHT87, GING87, RASH87, SZNY86]. The Mach operating system [RASH87] is one of the most recent systems to implement copy-on-write and is one of the few whose implementation has been published in detail. Copy-on-write is an integral part of Mach; it is the basis for both efficient message transmission and efficient process creation. This section briefly describes the Mach implementation of copy-on-write as it pertains to process creation.

A Mach process's address space is defined by an *address map* which is a linked list of references to memory objects. When a process forks, the memory objects are copied using copy-on-write. This is done by making the address maps of the parent and child point to the same memory objects. When a page in the copied memory object is written, a new page is given to the

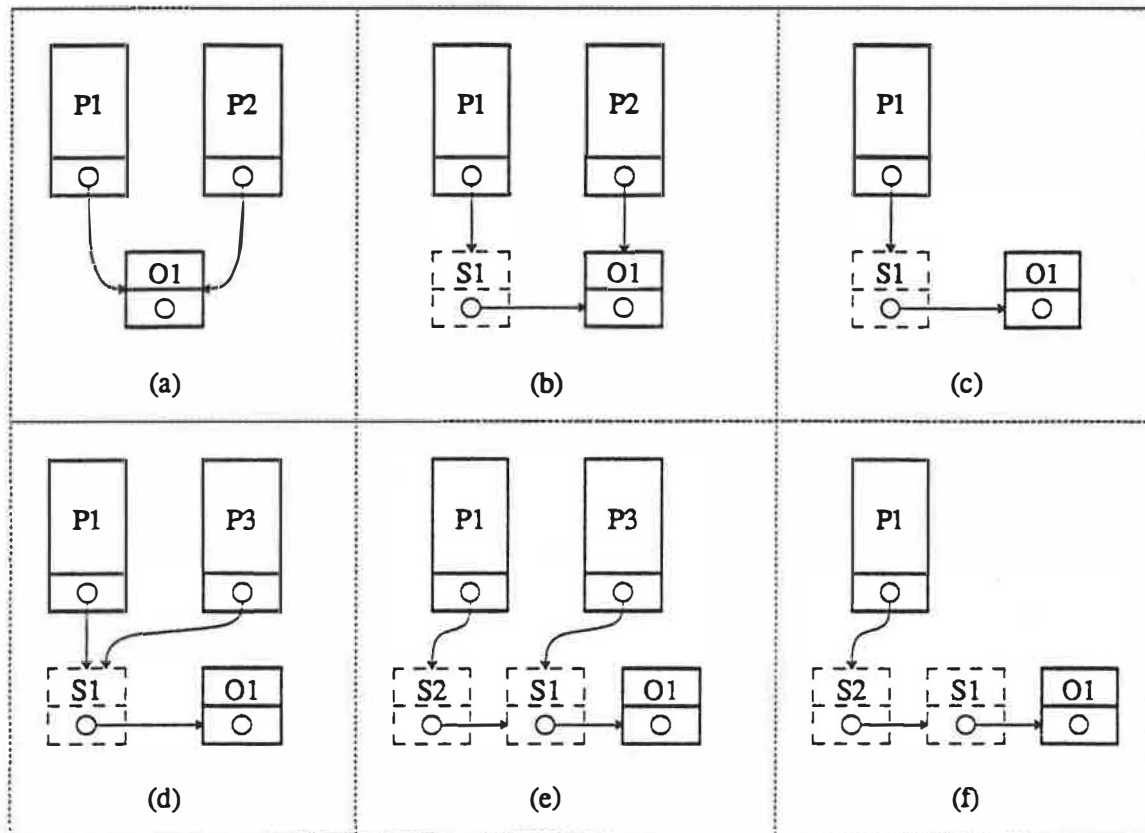


Figure 1. Mach copy-on-write. In (a) process P1 forks creating process P2. They both share object O1. In (b) P1 modifies a page and gets a shadow object to hold the modified page. P1's address map points to the shadow object which in turn points to the original object. In (c) P2 exits leaving P1 with the chain of two objects. In (d) P1 forks P3. They both share the object O1 and the shadow object S1. In (e) P1 modifies a page and gets a new shadow object S2. Now P2 has a chain of 3 objects: two shadow objects and the original object. When P3 exits in (f), P1 is still left with the 3 objects. However, by recognizing that the shadow objects completely overlap the original object, the extraneous shadow objects can be eliminated.

process that wrote the page. In order to hold new pages that are copied because of a copy-on-write fault, Mach creates an object called a *shadow object*; unmodified pages are kept in the original object and modified pages are kept in the shadow object.

The complexity that arises in the Mach scheme is that a shadow object may itself be shadowed as a result of a copy-on-write copy. This can result in an entire chain of shadow objects being created (see Figure 1). In order to satisfy a page fault, the list of shadow objects and then possibly the original object need to be searched to find the data for the page. Much of the complexity involved in Mach memory management is involved in preventing long chains of shadow objects [RASH87]. In particular, the extraneous shadow objects shown in Figure 1 that are left over after a child exits can be eliminated by recognizing that the shadow objects completely overlap the original object.

4. Sprite COW-COR

The Sprite copy-on-write scheme was designed with a more restrictive set of goals than Mach. The goals behind the Sprite design were:

- To make process creation efficient in a UNIX-like environment.
- To be able to run as efficiently as possible on machines such as SPUR that have virtually-addressed caches.
- To yield as simple an implementation as possible.

In particular, Sprite's copy-on-write scheme does not participate in the implementation of message communication; this simplified our design constraints in comparison to Mach.

4.1. Overview

Sprite uses a combination of copy-on-write and copy-on-reference, as illustrated in Figure 2. For each page that is involved in copy-on-write activity, one segment (called the *master segment*) has the page marked copy-on-write and all other segments that reference the page (called *slave segments*) have it marked copy-on-reference. When a process forks, the segment in the parent process becomes the master segment and the segment in the child becomes the slave segment. All pages that are resident or on backing store in the master segment are marked copy-on-write and made read-only. All pages in the slave segment that correspond to copy-on-write pages in the master are marked copy-on-reference and made inaccessible.

A copy-on-reference fault occurs when a copy-on-reference page is referenced. When the fault occurs, the master copy-on-write page is located, a copy is made, and the copy is given to the segment that contains the copy-on-reference page. In order to allow the master copy of the page to be easily located, the page table entry for each copy-on-reference page names the master segment for the page (as shown below, different pages may reference different master segments).

When a process attempts to modify a copy-on-write page (call it A), a copy-on-write fault occurs. A copy-on-write fault is more complex than a copy-on-reference fault because a new copy-on-write master segment for the page must be found; the original master segment will have its own writable copy of the page. This new copy-on-write master must be one of the slave segments. In order to allow a slave segment to be easily located, the master segment and each of its slave segments are linked together in a list; a master can have multiple slave segments if a parent forks multiple children. The new master segment is found by searching the list of segments for a slave segment that contains a page that is copy-on-reference off of A. This slave segment is given a copy-on-write copy of A (call it B). All of the remaining segments that have pages that were copy-on-reference off of A must now be changed to be copy-on-reference off of B. This is done by searching the list and updating the page table entries of each segment that was copy-on-reference off of A to point to the new master segment.

When a segment is deleted because a process exits or execs, copy-on-write dependencies in the deleted segment need to be eliminated. Pages that are copy-on-write must be copied to another segment. Each copy-on-write page (call it A) in the deleted segment is copied to another segment that contains a page (call it B) that is copy-on-reference off of A. If A is resident in memory this is done by remapping the page in A onto B. Otherwise the backing store for A is copied to B's backing store. Copy-on-reference pages in the deleted segment are ignored; this may cause extraneous copy-on-write page faults and is discussed below. Once all copy-on-write dependencies are eliminated, the segment is deleted from the linked list.

4.2. Trees of Descendants

The previous section only mentioned COW-COR for a single parent with multiple children. However, if processes that reference copy-on-write slave segments fork, then a tree of copy-on-

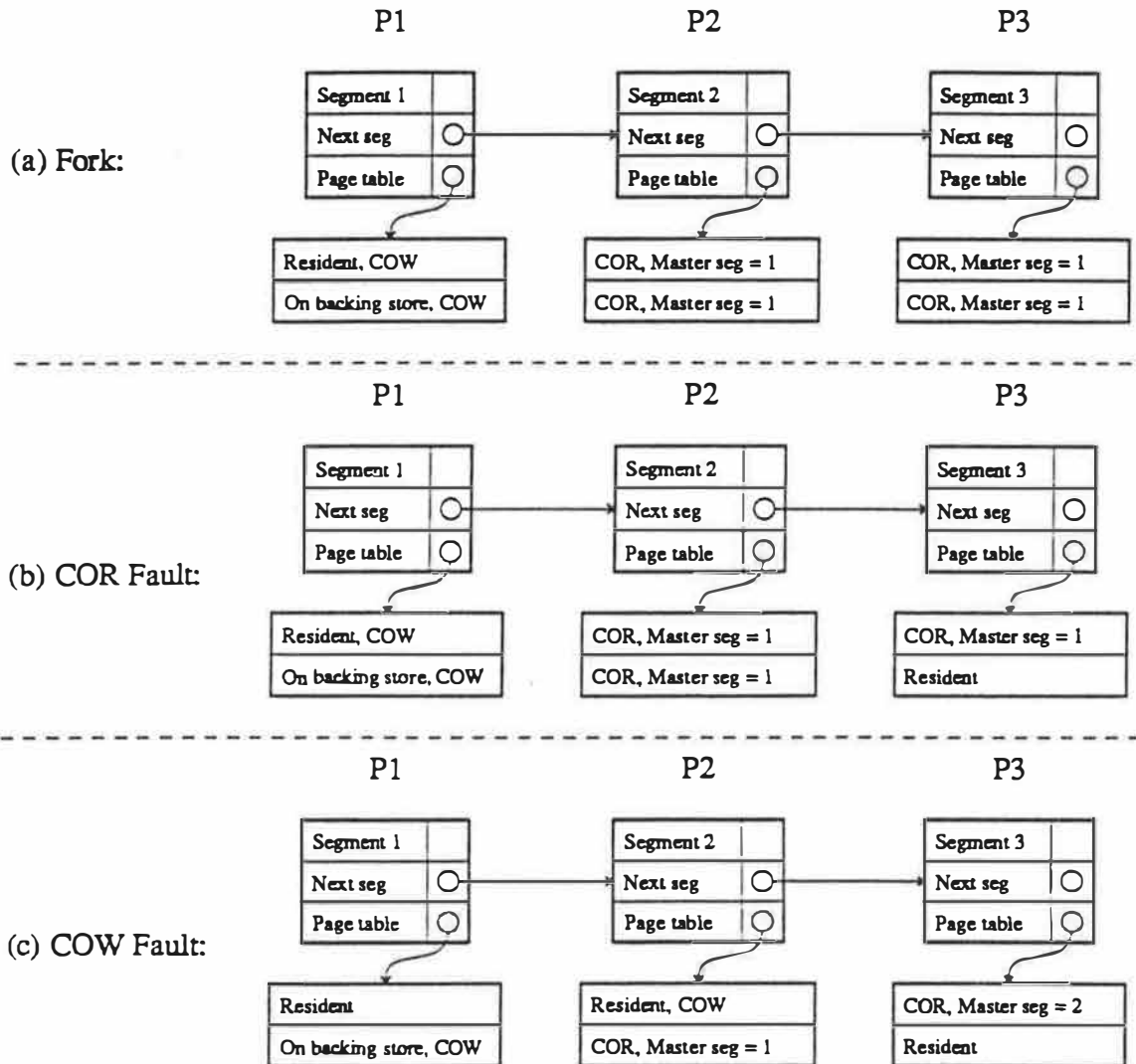


Figure 2. Sprite copy-on-write. In (a) the process (call it P1) that owns segment 1 forks two children and creates two copy-on-reference copies, segments 2 and 3, which are owned by processes P2 and P3 respectively. The page table entry (PTE) for each of the COR pages names the segment with the COW copy. In (b) P3 references the second page in segment 3 and a copy of the page is loaded into segment 3 from S1's swap file. The copy is made readable and writable. In (c) P1 modifies the first page in segment 1 and gives a new COW copy to segment 2. Segment 3's PTE is updated to point to segment 2 and segment 1's page is made readable and writable.

write and copy-on-reference relationships will result. Rather than build a tree-like data structure to represent the relationships, Sprite puts all of the related segments in the same linked list. This can be done because the page table entry for each copy-on-reference page names the segment that contains the master copy. This provides a simpler implementation and is based on the assumption that the lists will rarely contain more than a few segments.

One difference between the Sprite and Mach mechanisms is that when a page fault of any type occurs, the location of the master copy of the page is immediately known; no chain of objects needs to be traversed. However, Sprite does need to traverse its linked list of segments

for other reasons, as described above and below.

4.3. Eliminating Extra Copy-on-Write Faults

After a segment is deleted, or a copy-on-write fault or a copy-on-reference fault is handled, there can be pages marked copy-on-write for which there is no longer a corresponding copy-on-reference page. The easiest method of handling this problem is to cleanup extraneous copy-on-write pages when they are faulted on. However, because a copy-on-write fault is fairly expensive, the Sprite implementation of COW-COR checks for the common causes of extra faults and eliminates them.

One potential cause of extra page faults that is not handled is segment deletion. As explained above, copy-on-reference pages in a deleted segment are ignored, thus potentially leaving copy-on-write pages for which there is no copy-on-reference page. We chose to ignore this potential performance problem because we felt that it would be infrequent and it allowed us to simplify the implementation.

4.4. Backing Store

The backing store for each copy-on-write page is the master segment's backing store file. When a copy-on-reference fault occurs for a page that is on backing store and not resident in memory, the page is read from the master segment's backing store file. Copy-on-write faults can only occur to pages that are memory resident. If a process attempts to modify a copy-on-write page that is not memory resident, then a normal page fault occurs instead of a copy-on-write fault. Once the faulting process continues, it will try to modify the now-memory-resident page and a true copy-on-write fault will occur. This second fault could be eliminated by slightly complicating the implementation but the cost of the extra fault is very small in comparison to the cost of loading a page from backing store.

5. Comparison of Sprite Scheme and Shadow Objects

Besides using a combination of copy-on-write and copy-on-reference instead of pure copy-on-write, the major difference between Sprite and Mach is that Sprite does not use shadow objects. The method that Sprite uses to implement COW-COR could also be used to implement a pure copy-on-write scheme. The difference would be that when a process forks, each memory resident page would be marked copy-on-write in both the parent and the child segment's page tables, instead of copy-on-write in the parent and copy-on-reference in the child. For pages that are only resident on backing store, the page table entry of the child would be used to point to the parent segment since the parent has the swap file.

The main advantage of the Sprite method of implementation of copy-on-write is that it eliminates the potential to create chains of extraneous objects. For example, Figure 3 shows what happens under the Sprite scheme when a parent forks a child, the parent modifies a page, and then the child exits. The result is that after the child exits, the Sprite scheme automatically cleans up the list; there are no extra structures to maintain or collapse.

The disadvantage of the Sprite scheme is that it can require extra copies of pages when a parent exits before its child exits. With shadow objects, no copies are required when a process exits because shadow objects can exist even after the process that created them has exited. However, under the Sprite scheme when a segment is deleted all copy-on-write pages must be copied to another segment. In a normal UNIX environment parents usually wait for their children to exit, so in practice the Sprite scheme should perform as well as the shadow object scheme.

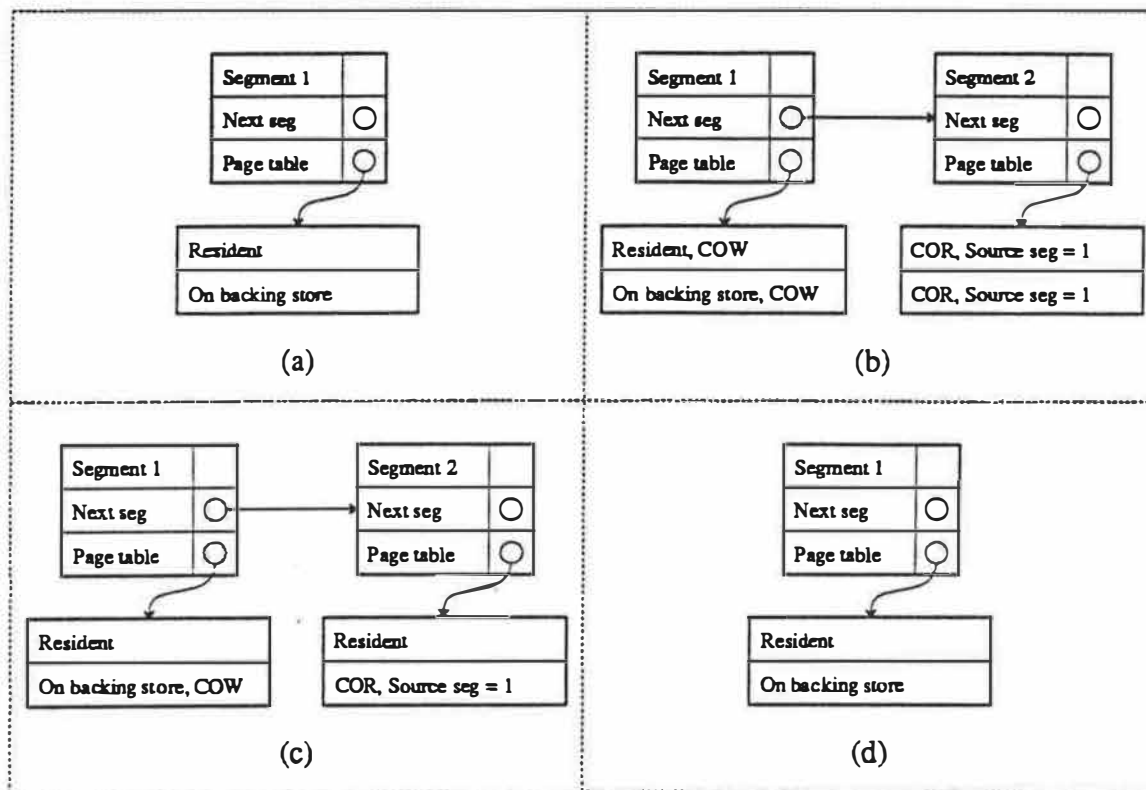


Figure 3. In (b) the process (call it P1) that owns segment 1 forks a child and creates a copy-on-reference copy, segment 2, which is owned by process P2. In (c) P1 modifies one of the copy-on-write pages in segment 1 and gives a copy of the page to segment 2. In (d) P2 exits causing segment 2 to be deleted. When segment 2 is deleted it is removed from the list and the lone copy-on-write page left in segment 1 is made readable and writable. The result is that the state of segment 1 is restored back to how it was in (a); that is, the state before segment 1 was created.

6. Copy-on-Write Performance

We ran benchmark programs and measured normal use of Sprite in order to answer several questions about the performance of the Sprite COW-COR scheme:

- What is the maximum potential benefit from COW-COR, compared to no copy-on-write mechanism at all?
- What is the actual benefit from COW-COR during normal use, compared to no copy-on-write mechanism at all?
- How does COW-COR compare to a pure copy-on-write scheme?
- How much more efficiently can COW-COR be implemented on SPUR than a pure copy-on-write scheme?

The benchmark programs are UNIX programs that have been converted to run on Sprite and the results obtained from the measurements of Sprite should be applicable to any UNIX-like operating system. The measurements were taken on a Sun-3/75 workstation with 16 Mbytes of memory, 8-Kbyte pages and about 2 MIPS processing power. The Sun-3/75 does not have a virtually-addressed cache.

6.1. Raw Performance

We used a simple benchmark to determine the maximum benefit attainable from COW-COR during process creation. This benchmark forks a child and then waits for the child to exit. The amount of memory that the parent has resident in memory or on backing store when it does the fork can be varied. It is an optimistic measurement of the benefit of copy-on-write because none of the pages are referenced or modified by the parent or the child. Table 1 gives the results.

There are two interesting results from this benchmark. First, it is slower to fork a process when all of its pages are memory resident than when all of its pages are on backing store. This is because the hardware protection must be changed to make memory resident pages copy-on-write. Second, forks are substantially faster under the COW-COR scheme than they are with copy-on-fork schemes: more than 10 times faster for processes with large amounts of resident memory and more than 100 times faster for processes with large amounts of memory on backing store. Thus, as expected, if processes with large amounts of memory fork and do not reference many pages, copy-on-write can substantially improve fork performance.

6.2. Realistic Performance

The benchmark described in the previous section gave a best-case scenario for the COW-COR mechanism: a large process forks and does not reference any of its memory. In order to make a more realistic determination of the benefits of the COW-COR mechanism over traditional copy-on-fork schemes, we measured a file system benchmark program, an edit-compile-debug benchmark and several day's work of two different Sprite designers. Table 2 describes the benchmarks and Table 3 gives the results.

One interesting result from Table 3 is that the number of copy-on-write pages is nearly equal to the number of copy-on-reference pages. This implies that in general there is only one segment that has any given page mapped COR. Therefore the COW-COR lists should normally contain only two segments and the extra overhead required to traverse the list on copy-on-write and copy-on-reference faults should be small.

Kbytes	COW-COR		Copy-on-Fork	
	Mem-res	Backing Store	Mem-res	Backing Store
0	22.8ms	22.8ms	22.5ms	22.4ms
64	24.7	24.0	59.7	171.7
128	25.8	24.3	79.6	265.0
256	28.0	24.6	119.4	457.6
512	32.3	25.3	199.0	850.8
1024	41.1	26.8	358.6	1635.1
2048	58.7	29.7	677.2	3209.0

Table 1. Raw Sprite COW-COR performance. This table gives the time required in milliseconds per execution of a *fork* and *wait* call in the parent and an *exit* call in the child as a function of segment size. These measurements were taken on a Sun-3/75. The first column gives the number of Kbytes that were either memory resident or on backing store when the parent forked. The second and third columns are the performance with COW-COR and the fourth and fifth columns are without COW-COR (i.e. all of the data had to be copied at fork time). "Mem-res" means that all of the bytes were memory resident and "Backing Store" means that all of the bytes were on backing store.

Benchmark	Description
Andrew	Copy a directory hierarchy containing 70 files and 200 Kbytes of data; examine the status of every file in the new subtree; read every byte of the files; compile and link the files. Developed by M. Satyanarayanan for benchmarking the Andrew file system; see [HOWA87] for details.
ECD	An edit-compile-debug benchmark run under the X11 window system.
User-A	Several day's work of a Sprite system designer using Emacs under the X11 window system. Work involved editing, compiling and other miscellaneous activities.
User-B	Several day's work of a Sprite system designer using typescript windows and a window-based editor under the X11 window system. Work involved editing, compiling, debugging and other miscellaneous activities.

Table 2. Sprite COW-COR benchmarks.

	COW Pages	COR Pages	Faults			Copies Saved
			COW	COR	% of Total	
Andrew	2846	2846	1%	71%	26%	28%
ECD	1430	1448	5%	72%	15%	22%
User-A	38771	40231	8%	63%	30%	28%
User-B	109965	112257	6%	66%	23%	27%

Table 3. Realistic Sprite COW-COR performance. This table gives Sprite COW-COR statistics for the two benchmarks and the two measurements of user activity. The second and third columns are the number of times a page was marked copy-on-write and copy-on-reference by processes forking. Columns four and five are the percentage of copy-on-write and copy-on-reference pages that actually generated faults. The sixth column gives the percentage of the total number of faults taken during the benchmark that were copy-on-write and copy-on-reference faults. Finally, the last column indicates how many page copies were saved by COW-COR relative to a copy-on-fork scheme.

Perhaps the most interesting result in Table 3 is that under normal use COW-COR saves less than 30% of the page copies that would be required under a copy-on-fork scheme. Furthermore, copy-on-write schemes require additional page faults that would not occur otherwise; as the cost of a page fault increases, the benefits of COW-COR will diminish. We determined from measurements of Sprite that a page fault takes 1.1 milliseconds. In addition, from Table 1 it can be calculated that the cost of a copy is approximately 2.5 milliseconds. Thus, in Sprite a page fault costs nearly half as much as a copy. Figure 4 shows that with this fault cost, COW-COR provides slightly worse performance than copy-on-fork.

The fault cost in Sprite is much higher than the fault cost in the Mach operating system [RASH88]. In Mach the page fault cost is less than 10% of the copy cost. If Sprite were able to

attain the same low fault cost as Mach, forks would be 15 to 20 percent faster with COW-COR than with copy-on-fork. Thus with a highly optimized implementation COW-COR can provide a moderate performance improvement over copy-on-fork schemes.

6.3. COW-COR vs. Pure Copy-on-Write

Nearly all of the faults that occurred during the benchmarks and normal use were copy-on-reference faults. If a pure copy-on-write mechanism could eliminate these faults, then it would provide much better performance than the COW-COR scheme. However, for the two benchmarks and normal use, between 80 and 90 percent of those pages that were copied because of copy-on-reference faults were eventually modified (see Table 4). Thus a pure copy-on-write scheme has only a small advantage over the COW-COR scheme: only between 10 and 20 percent of the page copies required under COW-COR would be eliminated.

Figure 4 shows the performance improvements possible on a Sun-3 with a pure copy-on-write scheme. With the high Sprite fault cost, a pure copy-on-write scheme provides a 5 to 20 percent improvement over copy-on-fork schemes and a 10 to 20 percent improvement over the Sprite COW-COR scheme. With the low Mach fault cost copy-on-write provides fairly substantial improvements over copy-on-fork schemes. Thus with an optimized fault handler, copy-on-write reduces the fork cost by from 30 to 40 percent over copy-on-fork schemes and by 10 to 20 percent over optimized COW-COR schemes.

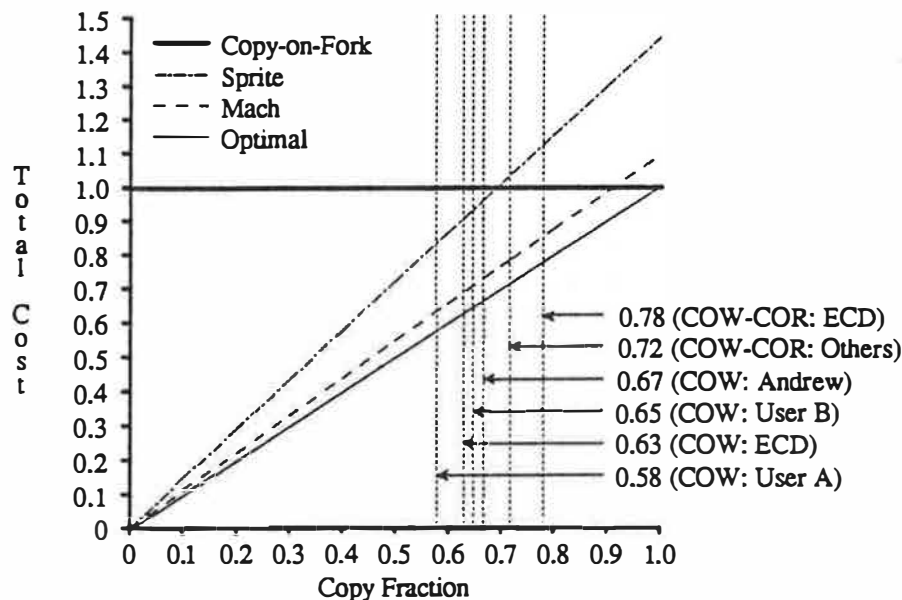


Figure 4. Total cost per page for Sun-3's as a function of the fraction of pages copied because of copy-on-write or copy-on-reference faults. A total cost of 1.0 corresponds to the cost of copying all pages at fork time. The optimal line represents the cost when the time required for each copy-on-write or copy-on-reference fault is 0, the Mach line when each fault is 0.234 milliseconds (9% of the cost of copying a page), and the Sprite line when each fault is 1.1 milliseconds (44% of the cost of copying a page). The 2 rightmost vertical lines correspond to the fraction of pages copied with the COW-COR mechanism for the benchmarks and the 4 leftmost vertical lines represent the fraction of pages that would have been copied with a pure copy-on-write mechanism.

	COW Faults	COR Faults	COR Modified	Pure-COW Faults
Andrew	1%	71%	93%	67%
ECD	5%	72%	81%	63%
User-A	8%	63%	79%	58%
User-B	6%	66%	90%	65%

Table 4. COW-COR vs. Copy-on-Write. This table gives the number of page faults that would occur under a pure copy-on-write scheme for the two benchmarks and the two measurements of user activity. The second and third columns show the percentage of copy-on-write and copy-on-reference pages that actually generated faults. The fourth column gives the percentage of those pages that were copied because of copy-on-reference faults that were eventually modified; all of the copy-on-reference pages that were eventually modified would have had to be copied under a pure copy-on-write scheme. The last column is the percentage of pages that would have been copied under a pure copy-on-write scheme; it is the second column added to the product of the third and fourth columns.

6.4. Cost of Virtually Addressed Caches

As mentioned earlier, one of the potential advantages of the Sprite COW-COR scheme over a pure copy-on-write scheme is that it may reduce overhead on architectures with virtually addressed caches, such as the Sun-3/200 and SPUR architectures. In these machines protection bits are stored along with the data in individual cache lines. To change the protection on a page, the operating system must first modify the page table entry, then flush all of the page's lines from the cache. When the lines are re-loaded into the cache, their protection bits will be set from the new page table entry.

When a process forks, all of its pages will have to be flushed from the cache in order to mark them read-only. This flush must occur in either a pure copy-on-write scheme or in Sprite's COW-COR scheme. In addition, whenever a copy-on-write page is made writable again, it will have to be flushed from the cache again. Once again, this will occur in both schemes. However, Sprite's mechanism allows a copy-on-reference page to be made accessible without any cache flushes: since the page was not previously accessible, there will be no data from it in the cache. On average, Sprite's COW-COR mechanism will require 2 flushes per page (one at the time of the fork and another one later when the parent's page eventually becomes writable again), while a pure copy-on-write scheme will require 2.6 (one at the time of the fork, another one when the parent's page becomes writable again, and a third one on the 58 to 67 percent of the pages that resulted in copy-on-write faults in the child).

The actual number of cache flushes required will be smaller on architectures such as SPUR that use direct-mapped caches. With direct-mapped caches the act of copying a page between virtual addresses that have the same offset within the cache will flush the source of the copy from the cache (the destination data will replace the source data in the cache because they will both map to the same address in the cache). Under COW-COR, over 70% of pages are copied. As a result an average of only 1.3 cache flushes per page will be required with a direct-mapped cache. A pure copy-on-write scheme copies over 60% of pages. This will give 1.4 cache flushes per page plus the 0.6 extra flushes explained above for a total average of 2.0 flushes per page.

Although Sprite's mechanism reduces cache flushing relative to pure copy-on-write schemes, the overhead may still be quite high. For example, if the cost of flushing a page is half as great as the cost of copying it, then any copy-on-write scheme will be at least as expensive as a copy-on-fork mechanism, even if none of the copied pages are ever accessed (unless the pages are on backing store). Since the Sprite COW-COR mechanism hasn't yet been ported to a machine

with a virtually addressed cache, we have no measurement of the impact of cache flushing on fork performance. However, we can estimate the impact of cache flushing on fork performance for the SPUR architecture. Because the actual performance on SPUR is dependent on numerous variables, including the cache miss ratio and the percentage of data that is modified, it is impossible to derive the exact fork cost on SPUR without actually measuring it. However, the worst and best case performance can be easily calculated.

Figure 5 gives the worst case and best case performance of copy-on-write and COW-COR on SPUR. The computation of the performance is a complex computation that involves the percentage of data that is resident in the cache during copy and flush operations and the percent of cache memory that is dirty (see Table 5 for the SPUR attributes that were used in the computation and Figure 5 for more explanation of the computation). In both the best and the worst case, COW-COR is strictly better than pure copy-on-write; this shows that COW-COR may be a reasonable alternative to pure copy-on-write for architectures like SPUR with virtually-addressed caches. In addition in the best case COW-COR and pure copy-on-write can cut the fork cost by a factor of 2 over copy-on-fork schemes, but in the worst case the fork cost is up to three times higher with COW-COR or pure copy-on-write. Thus although COW-COR and copy-on-write can potentially give a substantial performance improvement over copy-on-fork schemes, they can potentially give an even more substantial performance degradation.

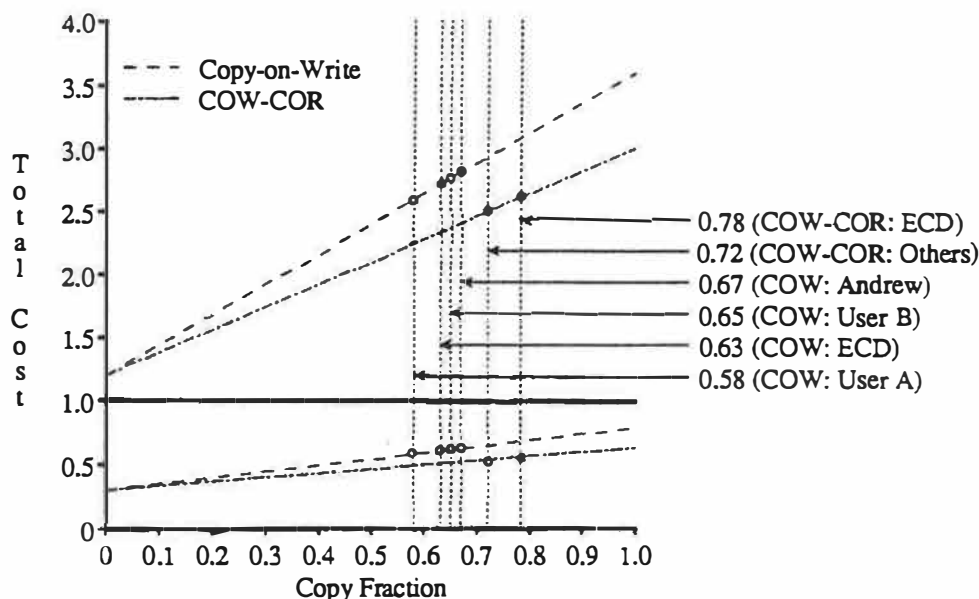


Figure 5. Total cost per page on SPUR as a function of the fraction of page copies and cache flushes because of copy-on-write or copy-on-reference faults. The attributes of the SPUR architecture that were used to compute the curves in the graph are given in Table 5. The lower lines of the graph are best case scenarios for copy-on-write and COW-COR and the upper two lines worst case scenarios. The best-case is the lowest possible flush cost and copy cost for copy-on-write and COW-COR and the highest possible copy cost for copy-on-fork. The worst-case is when all of the data for each page is present in the cache and clean at fork time and then the highest possible flush and copy costs occur for copy-on-write and COW-COR at other times. A cost of 1.0 corresponds to the cost of copying every page at fork time. The 6 vertical lines are the copy fractions for the 4 benchmarks.

SPUR Attributes		
Page Size		4096 Bytes
Cache Line Size		32 Bytes
Copy cost per cache line	Copy data	12 Cycles
	Cache read miss	23 Cycles
	Write-back data	22 Cycles
	Minimum cost	35 Cycles
	Maximum cost	80 Cycles
Flush Cost	Read tags	12 Cycles
	Flush clean line	9 Cycles
	Flush dirty line	25 Cycles
Page Fault Cost		500 Cycles

Table 5. Attributes of the SPUR architecture. When a cache line is copied at fork time or because of a copy-on-write or copy-on-reference fault, the destination of the copy will not be present in the cache. As a result the SPUR hardware will consider this a cache miss and fetch the destination from memory even though it is being totally overwritten. Hence the minimum copy cost is 35 cycles (12 for the copy and 23 to fetch the destination cache line). The maximum cost occurs when the source of the copy is not present in the cache (an additional 23 cycles) and the cache line that is being replaced must be written back (an additional 22 cycles). The cost of a cache flush ranges from 12 cycles if the line being flushed is not in the cache, to 37 cycles if the line being flushed is dirty. The fault cost is extrapolated from the Mach fault cost of approximately 500 instructions on a Sun-3 (234 microseconds on a 2 MIP machine). Since each instruction on SPUR takes one cycle, the fault cost is assumed to be 500 cycles. This fault cost is merely a rough estimate and will be higher when cache behavior is taken into account.

The main conclusion that can be drawn from this discussion is that copy-on-write mechanisms may not be worthwhile in architectures with virtually-addressed caches. The actual performance advantages of copy-on-write will depend on program behavior, the flush and copy costs of the architecture and on the actual implementation of the copy-on-write mechanism. Although Figure 5 was derived for the SPUR architecture and the Sprite implementation, a similar graph could be easily drawn to determine the potential benefit of copy-on-write mechanisms for any architecture and implementation.

6.5. Effect of Page Size

One reason why we measured only a small benefit from COW-COR under normal use may be that the measurements were made on a machine with large pages (8 Kbytes). If the page size were smaller, several changes in COW-COR behavior would occur:

- The total number of pages that are marked copy-on-write and copy-on-reference would increase.
- The total number of copy-on-write and copy-on-reference faults would increase.
- The percentage of copy-on-write and copy-on-reference pages that were faulted on out of the total number of copy-on-write and copy-on-reference pages would decrease.
- A pure copy-on-write scheme might improve relative to a COW-COR scheme because the percentage of pages that are copied on reference and then are later modified might decrease.

The only machine that we could perform our measurements on was a Sun-3. Although Sprite runs on Sun-2's, they do not have enough memory to allow us to measure either the ECD benchmark or normal use. However, since the trend in hardware is towards large page sizes, the results

that we measured on the Sun-3 architecture should be applicable to most machines that will be built in the future.

6.6. Effect on System Performance

In addition to just affecting fork performance, copy-on-write mechanisms can also potentially affect overall system performance relative to copy-on-fork schemes. First of all, by making forks faster, copy-on-write will improve overall system response time. However, since fork time may only account for a small portion of the execution time of a process, the actual improvement in overall system performance may be very small. For example the Andrew benchmark takes 280 seconds to complete. A pure copy-on-write scheme could eliminate 33% of the 2846 page copies (see Tables 3 and 4). If the fault cost is 0, then 2.5 milliseconds could be saved on each of the 939 page copies that would be eliminated, for a total savings of 2.3 out of the 280 seconds of execution time. This is less than a 1% improvement in the performance of the benchmark.

The other potential benefit from copy-on-write mechanisms is a reduction in memory use. By eliminating 30 to 40 percent of page copies that would have been required under copy-on-fork, the amount of memory required to fork a process with copy-on-write will be 30 to 40 percent smaller. If a very large process forks, then this can potentially result in a substantial reduction in the demand placed on physical memory which may result in an overall reduction in the number of page faults encountered by the system. However, for the programs that we measured the amount of memory saved by copy-on-write in relation to the amount of physical memory available should be insignificant. This is especially true given the fact that most processes immediately exec after forking so any extra memory will only be required for a brief instant. For example, in the Andrew benchmark no process that forks has more than about 150 Kbytes of stack and heap of which 100 Kbytes end up getting copied anyway. Since the machine that we ran the benchmark on has 16 Mbytes of memory, the 50 Kbytes that is unnecessarily copied is insignificant.

7. Conclusions

Copy-on-write has been gaining popularity in recent years as a mechanism to provide better fork performance. Our measurements of copy-on-write indicate that it can indeed provide a tremendous performance improvement over copy-on-fork schemes if very few of the virtually-copied pages are modified. However, our measurements of normal use indicate that more than 58% of pages that are shared copy-on-write do get modified. As a result less than 42% of the page copies that would have been required with a copy-on-fork scheme are eliminated. Thus although copy-on-write has tremendous potential, in practice it yields only a moderate performance gain over copy-on-fork.

There are two additional factors that may make it difficult to achieve even the 40% improvement suggested by the above measurements: page faults and cache flushing. An extra page fault will occur for each of the copy-on-write pages that ends up getting copied. Without a highly-tuned page fault handler, the additional page-fault overhead will more than compensate for the reduction in page copies. In addition, the architectural trend towards virtually-addressed caches has added the overhead of cache flushes to any copy-on-write implementation. The actual overhead will depend on the program behavior, the architecture, and the copy-on-write implementation, but our experience suggests that copy-on-write may actually result in worse performance than copy-on-fork for most applications on these machines. System designers need to pay very close attention to the fault cost and the cache flushing overhead if they wish to achieve the maximum benefit from copy-on-write.

The Sprite COW-COR scheme, which is a mixture of copy-on-write and copy-on-reference, provides a simple alternative implementation to pure copy-on-write schemes. It only requires 10 to 20 percent more page copies than a pure copy-on-write scheme, yet requires fewer cache

flushes on machines with virtually-addressed caches. Estimates of the cache flushing overhead on SPUR indicate that for the SPUR architecture COW-COR can actually provide slightly better fork performance than pure copy-on-write.

Acknowledgments

We are grateful to Rick Rashid for providing us with information about Mach, including the page fault cost, and for providing numerous helpful comments that helped improve the presentation of the paper. We also want to thank David Wood for helping us derive the cost of copy-on-write and COW-COR on SPUR.

8. References

- [AKHT87] Akhtar, P. "A Replacement for Berkeley Memory Management." *Proceedings of the USENIX 1987 Summer Conference*, June 1987, pp. 69-79.
- [BOBR 72] Bobrow, D.G., et al. "TENEX, a Paged Time Sharing System for the PDP-10." *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 135-143.
- [GING87] Gingell, R.A., Moran, J.P., Shannon, W.A. "Virtual Memory Architecture in SunOS." *Proceedings of the USENIX 1987 Summer Conference*, June 1987, pp. 81-94.
- [HILL86] Hill, M.D., et al. "Design Decisions in SPUR." *IEEE Computer*, Vol. 19, No. 11, November 1986, pp. 8-22.
- [HOWA87] Howard, J.H., et al. "Scale and Performance in a Distributed File System." *ACM Transactions*, Vol. 6, No. 1, February 1988, pp. 51-81.
- [MURP72] "Storage Organization and Management in TENEX." *Proceedings AFIPS Fall Joint Computer Conference*, Vol. 15, No. 3, 1972, pp. 23-32.
- [NELS86] Nelson, M. "Virtual Memory for the Sprite Operating System." Technical Report UCB/CSD 86/301, Computer Science Division (EECS), University of California, Berkeley, June 1986.
- [OUST88] J.K. Ousterhout, A.R. Cherenon, F. Douglass, M.N. Nelson, and B.B. Welch. "The Sprite Network Operating System." *IEEE Computer*, Vol. 21, No. 2, February 1988, pp. 23-36.
- [RASH87] Rashid, R, et al. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures." *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October 1987, pp. 31-39.
- [RASH88] Rashid, Richard, Personal communication.
- [SZNY86] Sznyter, E. W., Clancy, P. and Crossland, J. "A New Virtual-Memory Implementation for Unix." *Proceedings of the USENIX 1986 Summer Conference*, June 1986, pp. 81-88.

A Strategy for SMP ULTRIX¹

**Ursula Sinkewicz
ULTRIX Engineering Group
Digital Equipment Corporation
Nashua, New Hampshire 03062
sinkewic@decvax.dec.com**

ABSTRACT

Although UNIX² based symmetric multiprocessing systems are becoming more widespread, the software design varies considerably. Upon examining various systems, questions come to mind on the factors prompting each design. We explain how a symmetric multiprocessing strategy for ULTRIX evolved during the course of a prototype implementation. We stress the parts of the kernel that forced us to change our original approach.

1. PROTOTYPE OVERVIEW

Digital engineers recently wrote a prototype version of symmetric multiprocessor (SMP) ULTRIX. The prototype was developed by adding locks to the ULTRIX kernel. The locks protected global or shared data by allowing only one process to access the data at a time. The locks were based on VAX interlock instructions.

We started with the Asymmetric Multiprocessor (AMP) ULTRIX kernel,³ found the global data, and added locks to the data. We ended with a version of ULTRIX where:

- o Most system calls could run on any CPU in the machine.
- o Any CPU could handle interrupts from any device and from any other CPU.
- o All major kernel subsystems (virtual memory, process control, the network, the file system, and machine-dependent code) were fortified with locks so multiple processes could simultaneously access each subsystem.

2. LOCKING OVERVIEW

We devised a strategy, or set of guidelines, for deciding how much data to put under the control of each lock. We wanted lock granularity that promoted good system performance.

In part, our strategy was to begin with as few locks as possible and to add locks only when we could justify a need through measurement.

Generally, one lock protected large logically associated groups of data structures. We used lock contention measurements as a guideline for determining if the lock scope was reasonable. A typical result of high lock contention was noticeable system sluggishness.

¹ ULTRIX, MicroVAX, VAX, and DECnet are trademarks of Digital Equipment Corporation.

² UNIX is a registered trademark of A. T. & T. Information Systems.

³ See the TERMS section for definitions.

When lock contention was high, we reexamined the lock scope in hopes of deducing a better locking scheme. Our plan was to repeat measurements on the new locks to verify lower contention and improved performance. Unfortunately, we found no major cases where changing lock granularity would both improve performance and reduce contention. The locks that had severe contention problems were not reasonably adaptable to smaller granularity. In some cases we expected high contention and could have easily reduced granularity but were surprised to find low contention.

We finally reduced lock contention in isolated cases by recoding individual kernel routines. We changed, for example, process queue manipulation and noticed better performance.

The details of our experience working on the SMP ULTRIX prototype are presented below. We talk about decisions leading to our original strategy and how the strategy evolved during the course of the project. We emphasize the parts of the kernel that forced us to modify the original design.

3. BACKGROUND

The machines used in the prototype belong to the multiprocessors in the VAX 8000 family of Digital computers. We used a VAX 8200 modified to include 3 CPUs and a VAX 8300 modified to include 3 CPUs. Any CPU on the bus could receive interrupts from devices as well as other CPUs.

ULTRIX is a 4.2BSD based virtual memory system. It includes features such as DECnet, BISYNC, NFS,⁴ the Generic File System (GFS) and diskless support. ULTRIX runs on a wide variety of VAX machines from low end processors to servers.

4. EARLY DECISIONS

Our early decisions were mostly the result of a common sense approach made by people who had some previous experience with multiprocessing systems.

We decided to use locks as the mutual exclusion (mutex) mechanism. Mutex mechanisms are a means to prevent multiple processes from simultaneously accessing a shared resource such as global data. Locks allow one process exclusive access to a data structure. When a lock is successfully set by a process, another process cannot access the data until the lock is released. Releasing a lock causes one process to continue. Locks can either spin, waiting for a locked resource to become free (spin lock), or put a process to sleep (sleep lock).

We chose locks instead of other mutex mechanisms (monitors and semaphores) for design consistency. Sleep and wakeup are not used in the monoprocessor ULTRIX kernel as a mutex mechanism but they are used in notifying processes when events occur. A mutex mechanism is not necessary in monoprocessor ULTRIX because a process executing kernel code will retain control of the CPU until it is ready to relinquish control by, for example, lowering the processor's Interrupt Priority Level (IPL) to accept an interrupt, or sleeping. We decided to extend sleep and wakeup into a mutex mechanism. Locks do not replace sleep and wakeup but they do keep the flavor of event notification basic to sleep and wakeup.

A decision about locks was driven by the VAX hardware. The decision was to base the locks on the `bbssl/bbccl` VAX interlock instructions. These instructions atomically test and set a bit. Other interlock instructions are available on the VAX which can be used to make word updates and queue operations atomic. For example, any word aligned 16 bit field that needs locking may be changed with the VAX `adawl` interlock instruction. The VAX interlock instructions `insqhl` and `insqtl` make the operations of adding elements to the head

⁴ NFS is a registered trademark of Sun Microsystems Inc.

and tail of a queue atomic. We based our locks on the bbcci/bbssi instructions because they are cheap in tying up a bit instead of a word of memory. Bbcci/bbssi also let us fashion one lock primitive to update words, VAX longwords (32 bits), and queues. This freed us from having a variety of VAX interlock instructions in the kernel.

We decided to avoid recoding algorithms. Since our objective was to modify ULTRIX for SMP by adding locks, we did not want to obscure the objective with design changes to algorithms. Further, we did not want to have to debug algorithms and SMP problems at the same time.

A key design decision was to try to make debugging as easy as possible. We fortified the lock primitives with error detection. We added a lock hierarchy and a timeout feature to spin locks.

Lock hierarchy prevents the general situation where process A locks resource 1, process B locks resource 2, then processes A and B each compete for the resource the other process holds. The situation is deadlocked because the resource each process needs is locked by the other process. Ordering the locks or establishing a lock hierarchy, eliminates the deadlock by forcing both processes to contend for one resource first.

The timeout feature in spin locks helps detect the coding error of locking a resource and then forgetting to unlock the resource. Now, if another process tries to lock the resource with a spin lock, it will timeout.

SMP can be achieved by two extremes. One extreme is adding one lock to the kernel allowing at most one process to use kernel data at any time. Another extreme is adding a myriad of locks protecting each individual structure. Both extremes should lead to highly suboptimal performance.

We wanted to end with the minimum number of locks needed to (1). maintain data integrity on a multiprocessor and (2). maximize performance gain. One approach was to start with a small number of locks with each lock controlling large amounts of data, and decompose into more locks controlling smaller quantities of data when measurement showed high contention. An alternate approach was to start with a lot of locks and then try to reduce the number of locks by increasing the scope of selected locks, in order to improve performance. We decided to use the first approach because it seemed easier.

We agreed to try not to hold locks across routine calls. For good performance, locks should be held for the shortest possible time. We thought that consistently trying to release locks before exiting routines would help eliminate cases where we might hold a lock longer than absolutely necessary.

A key decision we made early on was when we would stop working on the prototype. It is quite easy to add locks to a kernel and see a significant performance loss. It is difficult to decide on optimal locking schemes resulting in substantial performance improvement. We established two performance goals at the start of the project. One goal was that the SMP kernel, when running on a uniprocessor, would degrade performance by no more than 3% from a similarly configured AMP kernel. The 3% was our estimation of lock overhead.

The second performance goal was to improve the performance on a 2 CPU machine running SMP ULTRIX by 1.5 times a similarly configured ULTRIX kernel running on a uniprocessor. The amount of software performance improvement one can expect given a number of processors is an issue. Our estimates are what we saw as reasonable given this was a prototype implementation.

Our early design decisions led us to a set of coding rules we call a strategy. Several design decisions including ease of debugging were reflected in our lock primitives. The strategy at the beginning of the project was:

- o Try not to hold locks across routine boundaries.

- o Use the largest reasonable lock granularity for the subsystem.⁵
- o Prove the need for more locks by performance or lock contention measurements.
- o Do not recode algorithms.

5. SELECTING WHAT TO LOCK

The first question we needed to answer before applying the strategy was, "What should be locked?". The most general answer is to lock a subset of shared kernel data. Protecting shared data prevents two processes running kernel code, on two separate CPUs, from simultaneously accessing the same field in the same structure. If this happens with a counter update, for example, it could mean one instance of the entity being counted is lost.

The notions of locking a resource, setting a lock, and locking a data structure, are synonymous. We added fields containing lock information to selected data structures. Other structures were protected by locks outside the structure. These locks were used as a key where one key locked all of a particular resource. Any process trying to get an mbuf, tried to get the mbuf "key" or lock. If another process happened to be using the "key", then the intervening process delayed.

The locks come into play when a process tries to update one of the selected global structures. The process first tries to set the lock by testing an interlock bit. If the interlock bit is set, then the resource is considered to be in use. If the bit is not set, then the resource is considered to be free. The process making the test can either spin by testing the interlock bit until it is free, or yield to some other process.

We began the ULTRIX prototype work with a detailed code inspection. We charted the procedure calls in each kernel subsystem. The point was to see where data was updated, how it was updated, and to identify where and which data structures interact between subsystems. Interaction between subsystems was important in organizing the lock hierarchy.

The kernel data can be grouped into the categories (1). write once read many (WORM), (2). write many read once (WMRO), (3). private data, (4). don't care, (5). local communication, (6). divisible, and (7). critical shared data.

WORM data does not need locking. This data is written once and referenced repeatedly. An example of WORM is the system pages mapping kernel text. This mapping is made at bootstrap time and does not change while the system is running.

Most WMRO data does not need locks. On VAX longword aligned writes of 32 bit fields, the VAX guarantees that one of several simultaneous writes will be successful. If the ensuing read is fairly random then accuracy may not be critical and a lock is not necessary. This might be the case when collecting some status information. An example of WMRO is runrun, a scheduling request flag, which takes on binary values. Runrun can be set several times before its value is read and acted upon. Note that WMRO data which is not the same size as a VAX longword may require a lock.

Private data, such as fields in a process' u area, needs no locks.

Don't care data, such as access permissions in the credentials structure used in filesystem operations, needs no locks. When accessing a file on a remotely mounted file system, local and remote access permissions are checked automatically. The access permissions can change between being checked locally and remotely. Ensuring that access permissions do not change is unnecessary because the ensuing access to the remote file, read or write, will succeed or fail according to the permissions on the file at the time of

⁵ The subsystems are virtual memory, process control, the network, the file system, and machine-dependent code.

access.

Data used for local communication needs no locks. An example is the variable `tcpcksum` which is used exclusively in the TCP routine processing incoming data. The variable is global yet it is used as a local variable. The SMP solution is not to lock `tcpcksum` but to make it local to the routine using it.

Divisible data needs no locks. Divisible data is either a structure containing shared and local information, or data that can be associated with a CPU. Divisible data can be separated into truly shared data and one of the alternatives listed above. Only the shared part needs a lock. It is not necessary to set the lock when updating any field in the structure. It is only necessary to set the lock when updating the truly shared elements in the structure.

The way divisible data is updated is important because some minor change, namely associating the data with a CPU, can eliminate the need for a lock. Consider statistics updates, for example. The structure `mbstat` is used to track mbuf information. On an SMP system, many processes can be updating the `mbstat` fields simultaneously and we want accurate data, so `mbstat` needs protection from simultaneous updates.

We can divide the `mbstat` updates among the CPUs. Instead of using one `mbstat` structure, we can use several where each structure is associated with a CPU. A process running on CPU A will update the `mbstat` structure for CPU A. A lock will not be necessary because only one process can be running on CPU A at a time. Applications can collect data from the individual instances of `mbstat` for each processor.

Generally, the data left needs locks. This is the critical shared data.

In summary, our prototype approach to the elusive question of "What data needs a lock?" began with a detailed examination of each kernel subsystem. The point of this study was to see where the subsystems interact, how the data is used, and what data is shared. Once the shared data is identified, then a question needs to be asked of each structure. "Does the data belong to one of the categories listed above?" If the answer is yes, then the data can be discounted. The remaining shared structures need locks.

6. FIRST PASS RESULTS

We staged our SMP work to go from an AMP to an SMP system on a per system call basis. Once we had added locks to all routines in a particular system call, we changed the data structure `sysent[]` to indicate the particular call could run on any processor. Before entering the system call, the kernel checked `sysent[]` to see if the call required a switch to the primary CPU.

We measured contention with an application that read runtime lock statistics. Our lock structures were augmented with statistics fields. We kept track of the number of times the lock was used, the number of times processes tried to set the lock but failed, and the number of times the lock was successfully set. Our contention measurements were basically a runtime accumulation of these numbers for each lock in the system. Figure 1 shows a sample of the type of information we collected. Note that the numbers are in decimal.

LOCK	WON	LOST	SPUN	AVGSPIN	%LOST
<code>actv</code>	0	0	0	0.000	0.000
<code>bio</code>	2476812	809	8718	10.776	0.033
<code>buckets</code>	703999	0	0	0.000	0.000
<code>timeout</code>	19259427	3762	39526	10.507	0.020

FIGURE 1 Lock Contention Measurements

Parts of the kernel adapted ideally to our SMP strategy. For example, we used one lock to control resource maps (`rmfree`, `rmalloc`, etc.). We expected high contention on the lock because of the number of maps the lock was controlling. However, we saw low lock contention and had no need to change the lock granularity.

Other parts of the kernel did not adapt to our strategy as well. We saw instances of very high lock contention in process queue manipulation and mbufs. We saw instances where lock contention was low but we were plagued with bugs indicating SMP errors had occurred.

7. PROBLEMS AND FIXES

Our problems were either high lock contention or SMP bugs. SMP bugs were mostly coding errors, indicating missed wakeups and unprotected shared data.

7.1. High Contention

When lock contention was high, we reexamined the relevant data to see if we could alleviate the contention problem by adding another lock. In all cases where the data under one lock could cleanly be divided and brought under multiple locks, we saw low contention. In the cases where we had severe contention, we saw no solution but algorithm change. The major cases were process control and the system time variable. We focus on the system time variable here.

7.1.1. The System Time Variable

The system time variable consists of 2 VAX longwords. One longword is the seconds field and the other is the microseconds field. The time variable is read by device drivers, network routines, file system routines, and the system call `gettimeofday()`. It is written by bootstrap routines, `hardclock` and the `settimeofday()` system call.

On a single CPU machine, ULTRIX guarantees that both fields in the time variable are updated without interruption by elevating IPL. This prevents inaccurate time because the IPL level prevents a delay from taking a device interrupt between updating the microsecond and second fields.

Elevating IPL on an SMP system prevents device drivers from interrupting an update on one CPU, but does not prevent processes on different CPUs from interfering with the system time variable in the middle of an update.

Consider what might happen when the second and microsecond fields are not updated atomically on an SMP system. Say the current system time variable holds the value 5 seconds and 3 microsecond and a process on CPU A is about to update it to 6 seconds, 0 microseconds. There is a process on CPU B reading the system time variable. It sees 3 in the microsecond field. Say the process on CPU A now writes 0 to the microsecond field as part of updating to the new value. Then the process on CPU B continues and reads the second field which is still 5. Then say a process on CPU B makes another read of both fields before CPU A's process can update the second field.

CPU B's process first reads 5 seconds and 3 microseconds in the system time variable. Its second read is 5 seconds and 0 microseconds. The process on CPU B is seeing time run backwards!

A lock guaranteeing an atomic update to the microsecond and second fields solves the problem of time running backwards. However, locking substantially slows the system because of the time needed to set and release the lock. Further, competition for the lock is high because the system time variable is updated 100 times per second and read often.

Our solution was to change the way the microsecond and second fields were updated. We created both a copy of the system time variable and a pointer to the system time variable. The pointer always referenced either the copy or the original variable. On a write, the

copy might be updated. After the write is completed, the pointer is changed to reference the updated time. On a read, the pointer is read first, then the longwords it points to are accessed.

Note that an algorithm that checks the value of the system time variable several times before doing a write, still has the problem of time running backwards. An algorithm can store the system time variable locally, compare the local values with the actual values and loop if the values do not match. This does not prevent the situation where time runs backwards because the write done by one process can be interrupted and delayed. Any number of reads can be done during the delay and the same situation described above can occur.

Our solution may result in the process on CPU B reading old values in the system time variable but the process on CPU B will never see time run backwards (a new microsecond field and an old second field).

This solution to the problem of fast quadword updates changed the way the system time variable was accessed and how it was stored. We found the best solution was to avoid locks even though we were trying to adapt the system time variable for SMP by adding locks.

A better SMP strategy allows changing algorithms when necessary. The decision to recode should be made after lock schemes are measured for contention because it is difficult to predict lock contention. We saw high contention on the the system time variable lock but were surprised with low contention on the resource map lock.

7.2. SMP Bugs

We found SMP bugs in three areas. They were (1). a missed wakeup problem, (2). holding locks across routine calls, and (3). coding errors.

7.2.1. Missed Wakeups and Lost Resources

We found missed wakeup problems to be annoyingly persistent. An example of missed wakeup was seen when locks were added to code already containing sleep or wakeup. We saw bugs in some unlock, sleep, lock sequences. Between doing the unlock and sleep, another process did a lock, changed the resource, and issued a wakeup. We needed to release the lock before going to sleep because the process doing the wakeup also needed the lock. By the time the first process got to sleep, it had either missed the wakeup it was expecting or the resource had disappeared.

We had taken pains to eliminate the missed wakeup problem in the lock primitives. We were aware that this type of situation might occur in some existing sleeps and wakeups in the kernel. However, the problem occurred infrequently and sporadically. The missed wakeup problem occurred only when the state of the resource was critical to the wakeup. Mbufs used sleeps and wakeups often yet we never saw a missed wakeup problem here. This was because mbuf wakeups occur whenever an mbuf is freed. Mbufs are freed continuously so one missed wakeup does little or no damage. Areas where we saw the missed wakeup problem were the device drivers and the file system.

Our fix to the missed wakeup problem was to replace particular instances of sleep with a version of the system call we had modified for SMP. Our modified sleep was used when it was critical a wakeup was not missed. Modified sleep put the process on the sleep queue and then released the lock. This prevented another process from grabbing the resource between the unlock and sleep operations.

7.2.2. Locks Across Routine Calls

Our strategy of trying not to hold locks over routine calls was very hard to apply to the network. We found it necessary to add excessive validity checking at the beginning of routines. This checking was to ensure, for example, socket pointers and protocol control block

pointers had not been deleted by another process in the time between relinquishing and resetting the lock. Because we tried to relinquish locks at the end of routines, we expected to add an amount of validity checking, especially on pointers, at the beginning of routines. Even with a fair amount of validity checks we still found indeterminate behavior and very slow operation. We believe this was caused by odd combinations of socket, TCP, and IP data caused by different processes manipulating the same socket.

Our strategy of trying not to hold locks over routine boundaries loosely associated lock placement with routine boundaries. This is something we had to ignore entirely in the network code. We changed our strategy to simply hold locks for as short a time as possible.

The lock scheme we used in the network remained centered on the socket. We associated a lock with the socket structure. Each successful socket call created a unique lock. We changed the lock operation to set the socket lock early (namely in the `screate()`, `sosend()`, etc. set of kernel routines). The lock was held until most of the work for the particular operation was finished. For example, when creating a socket, we set the lock as soon as the kernel had allocated space for the socket and held it until all initialization and setup was complete for the call. This lock scheme eliminated all checking to ensure data consistency in the beginning of routines.

7.2.3. Coding Errors

There were cases where we overlooked data that should have been protected with a lock. There were cases where we forgot to match a lock operation with an unlock operation. We changed our lock primitives to check that unlocks were preceded with locks. Over a period of time, both of these instances of coding error were flagged by the lock primitives and fixed.

We considered writing a preprocessor to inspect the files for coding errors, such as unmatched lock and unlock operations. We rejected the idea because of the time needed to write a preprocessor that had adequate capabilities. Since we were holding locks across routine boundaries, the task of matching lock and unlock operations with a preprocessor became complicated. Once the preprocessor found a lock operation, it would have had to verify that all possible nested routines did the appropriate unlock operation.

We found improving the lock primitives to be an easy way of exposing a reasonable variety of SMP coding errors.

8. FINAL STRATEGY

Generally, we found it very useful to be conservative about adding locks. Frequently, we predicted a need for more locks but were proved wrong by low runtime contention measurements.

We tried to use routine boundaries as a flag prompting us to consider releasing a lock. We found it absolutely necessary to hold locks across routine boundaries, but were continuously asking ourselves if the lock could be freed. In the network code, however, we had to ignore routine boundaries entirely.

• The strategy that evolved as we did the work became:

- o Hold locks for as short a time as possible.
- o Use the largest reasonable lock granularity for the subsystem.
- o Prove the need for more locks.
- o Recode algorithms only when lock contention is high and more locks do not solve the problem.

9. CONCLUSIONS

Our prototype took about 3 months of earnest development time. Five of us worked on the project. We found physical proximity essential as we needed to talk with each other every day.

We generally met our performance goals. The SMP kernel running on a uniprocessor ran slightly slower than an ULTRIX kernel without locks. The SMP kernel running on a 2 CPU machine was about 1.5 times faster than the kernel running on a uniprocessor.

Our approach was not hardware dependent. Noticeably bad performance always indicated a lock problem. We used bad performance as a sign, telling us it was time to take another set of lock contention measurements.

We cannot overemphasize the importance of stress testing. The easy SMP problems of missing a lock now and then can be picked up by lock primitives. The obscure problems indicating improper synchronization or some other SMP type problem surfaced when the subsystems were exercised.

The deadlock detection in the lock primitives was invaluable. Of course, this was at the expense of kernel size and kernel performance. Kernel size can be reduced by using abbreviated lock primitives that can be compiled conditionally when say a debug switch is off.

Measuring lock contention was extremely important because we could not accurately predict contention on a lock. We found it useful to run tests on a regular basis. We found many instances where one improvement broke another.

Overall, it was possible and reasonable to perturb the kernel with locks. It was possible to measure the kernel's behavior, isolate the poor performance, and make improvements. We found this iterative approach to be a successful way of modifying a kernel for SMP.

10. TERMS

Asymmetric Multiprocessing (AMP) – Refers to a computer system that has at least two processors where work is not democratically divided among the processors. This typically refers to a system explored at Purdue University where all processes requiring system service are context switched to the primary processor for execution. The primary processor typically services all device interrupts. The secondary processor typically satisfies user requests which on a VAX, are made in User Mode.

Atomic Operation – A sequence of computer instructions that cannot be interrupted.

Contention – Competition for a lock. High contention shows many processes attempting to set a lock but failing.

Deadlock – A term describing a situation where two (or more) processes both hold a resource and each process waits for the resource held by the other process. Neither process can proceed to release its resource because it is waiting for the other processes' resource to be freed.

Interlock instruction – A machine instruction which guarantees, through hardware, singular access to a particular memory location.

Lock Granularity – Refers to an amount of data controlled by a lock. Low granularity implies a lot of data is under control of the lock. High granularity means a small amount of data is under control of the lock.

Lock Scope – The fields in one or many data structures under the control of a particular lock.

Monitor – A method for controlling access to both re-entrant code and shared data.

Mutex Mechanism – A mechanism preventing multiple processes from simultaneously accessing either code or data.

Semaphore – A mutex mechanism that involves operations using resources working in conjunction with operations freeing resources. Semaphores can queue multiple requests.

Sleep lock – A lock which, upon finding a lock set, puts the associated process to sleep. Sleep locks cannot be used at interrupt level.

Spin lock – A lock which continuously tests an interlock bit. The test continues until the lock is able to set the interlock bit. Spin locks can be used at interrupt level.

Symmetric Multiprocessing – A system containing multiple CPUs where most activity on the system is equitably distributed among all the CPUs. It is a democratic situation where no particular CPU is favored for work. There are degrees of symmetric multiprocessing. Systems that boot to one CPU can still be considered SMP systems if all CPUs are treated equally on a runtime basis.

11. BIBLIOGRAPHY

M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems", AT&T Bell Laboratories Technical Journal, Volume 63, Number 8, October 1984.

Stuart Farnham, Michael S. Harvey, Kathleen D. Morse, "VMS Multiprocessing on the VAX 8800 System", Digital Technical Journal, Number 4, February 1987.

G. H. Goble, "A Dual Processor VAX 11/780", Usenix Conference Proceedings, September 1981.

Jack Inman, "Implementing Loosely Coupled Functions on Tightly Coupled Engines", Usenix Conference Proceedings, June 1985.

M. D. Janssens, J. K. Annot, and A. J. Van De Goor, "Adapting UNIX for a Multiprocessor Environment", Proceedings of the ACM, Volume 29 Number 8, September 1986.

12. ACKNOWLEDGEMENTS

The people who worked on the prototype deserve recognition because without their efforts, there would be no story to tell. Miche Baker-Harvey was project leader and was also responsible for process control, the general design, and the debugger. Joe Martin did all the virtual memory work. Jim Woodward made all hardware specific changes. Jeff Chase worked on the file system. The author made the network changes.

A Heap-based Callout Implementation to Meet Real-Time Needs

Ronald E. Barkley
AT&T Information Systems
Summit, New Jersey
attunix!rebark

T. Paul Lee
AT&T Bell Laboratories
Holmdel, New Jersey
hocus!tpl

ABSTRACT

The UNIX® operating system provides support for the real time scheduling of events through a mechanism known as the *callout* table. To meet the needs of real-time applications, the execution times of the callout scheduling operations must be short with small variance. Unfortunately, the callout table has traditionally been implemented as a linearly ordered array, and it performs poorly when heavily used. In this paper, we describe the design, analysis, and measurements of a callout table based on a heap data structure. Measurements taken on a prototype built with UNIX System V Release 2 suggest that a heap-based callout table is much more efficient and predictable than the standard array-based table, and thus is better suited to meet the requirements of real-time applications. We also discuss a delayed event heuristic that buffers events without reheaping; this strategy can cut in half the cost of some callout operations.

The current implementation of the callout table uses *relative* timing to sequence events; in this paper we discuss using *absolute* timing in the table to better capture the essence of event ordering and allow implementations that use new data structures. We also analyze a linked list-based callout table and compare its amortized computational complexity to that of the heap-based table.

1. INTRODUCTION

The UNIX® operating system^[1] manages many physical and logical resources and must coordinate and schedule different events on their behalf. To support some kernel services, particularly physical devices and network protocols, certain kernel functions must be called on a real time schedule. In essence, each service requires its own interval timer, and the kernel has to provide a timing mechanism for all the services it manages; this mechanism is implemented through the *callout* table. Furthermore, to meet the needs of real-time applications, the callout functionality is extended to user level processes. Real-time processes can make heavy demands on the callout table, but to guarantee real-time response they require that the execution times of the callout operations be short with small variance. In this paper, we describe an implementation of the callout table based on a heap data structure; this implementation provides inexpensive and predictable costs for callout operations, especially when the callout table is being heavily used by real-time applications.

Callout services are available through two functions: *timeout()* and *untimeout()*. A kernel subsystem or real-time application can request that a particular function be invoked after a certain amount of real time has elapsed by calling the *timeout()* function. *Timeout()* is passed three parameters: an interval in ticks¹,

the function to be invoked after that interval elapses, and the argument that the function uses. *Timeout()* schedules a call to the function by placing the relevant information in the callout table, and returns a unique identifier. This identifier can be used later in a call to *untimeout()*, which revokes the earlier *timeout()* request. For example, a network protocol routine will revoke the watch-dog *timeout()* call when it proceeds without error. Internal to the callout mechanism is an additional function *timein()* that actually calls the requested functions (with their given argument) when the prescribed interval expires.

Traditionally, the callout table has been implemented as a linearly ordered array, and the array-based callout table is known to perform poorly when heavily used; in Section 2 we analyze its performance characteristics. In Section 3, we describe the linked list-based callout table, which is a slight improvement over the array-based version. In Section 4, we present the heap-based callout table and analyze its theoretical and amortized performance characteristics. Controlled experiments were conducted with a prototype of the heap-based callout table; these empirical results are presented in Section 5. We then introduce a delayed event heuristic that further cuts down the cost of callout operations. In the last section, we summarize the work.

2. ANALYSIS OF AN ARRAY-BASED CALLOUT TABLE

2.1 An Overview

In the array-based implementation, the kernel keeps the entries in the callout table sorted according to their respective "time to fire," independent of the order in which they were placed in the table^[2]. Because of this time ordering, the time field for each entry is stored as the amount of time that must lapse after the previous entry fires. Thus, the total time to fire for a given entry is the sum of the times to fire of all entries up to and including the entry itself. At every clock tick, the kernel decrements only the "time to fire" counter of the *first* entry of the callout table. We refer to this arrangement of times as *relative timing*.

In this implementation, when *timeout()* is called, we do a linear search through the array to find the proper location to insert the new entry. To fit the entry in, we have to expand the array at the insertion point by shifting all the entries down a slot. When *untimeout()* is called, we conduct a linear search looking for the given identifier; if it is found, we remove the corresponding entry and then compact the rest of the array after the deletion point. In both cases, we must adjust the relative timing of the entries around the insertion or deletion point. When the internal *timein()* routine is executed, we remove the first entry of the callout table, and then compact the entire table.

2.2 Computational Complexity

The computational complexity^[3] for these three operations in an array-based implementation is summarized in Table 1. To illustrate the details of the data structures and algorithms, each operation is subdivided into two phases, *search* and *reorganize*. Except for the constant search time for *timein()*, all operations are linear with respect to *n*, the number of entries in the callout table.

TABLE 1. Computational Complexity for an Array-Based Callout Table

Operation	Search	Reorganize
<i>Timeout()</i>	$O(n)$	$O(n)$
<i>Untimeout()</i>	$O(n)$	$O(n)$
<i>Timein()</i>	$O(1)$	$O(n)$

3. AN IMPROVED VERSION BASED ON LINKED LISTS

In some UNIX implementations where the callout mechanism is exercised frequently (especially on mainframes), a linked list-based callout table is used; the linked list avoids array compaction and

1. A tick is the minimum internal clock resolution; for minicomputers, it is typically in the order of 10 milliseconds.

expansion. The re-organization complexity is improved over that of the array-based implementation but at the cost of adding explicit links between the callout table entries. The computational complexity for this implementation is shown in Table 2.

TABLE 2. Computational Complexity for a Linked List-Based Callout Table

Operation	Search	Re-organize
<i>Timeout()</i>	$O(n)$	$O(1)$
<i>Untimeout()</i>	$O(n)$	$O(1)$
<i>Timein()</i>	$O(1)$	$O(1)$

The linked list-based implementation still uses relative timing. Given the way the kernel clock routine updates the callout table, *relative* timing seems clever since it eliminates the need to decrement all the time entries in the callout table. However, if we examine the need of timekeeping in the context of event scheduling and de-scheduling, the relative timing is an artifact of array-based and list-based implementations.

Using *absolute* timing in the callout table is both conceptually straightforward and easy to implement. The kernel already maintains a variable (usually called *lbolt*) that counts the number of ticks since the most recent system reboot. When *timeout()* is called, we can schedule the event in a future time epoch by adding the interval parameter given in the *timeout()* call to the current value of the *lbolt* variable. When *lbolt* reaches this value, it is time to schedule the event. Absolute timing eliminates the need to decrement time entries in the callout table, and also the need to maintain relative timing when events are inserted and deleted from the table. Finally, absolute timing allows us to use a wider range of data structures since we are no longer constrained by the relative ordering of events in the callout table.

The heap-based callout table is an alternative implementation that uses absolute timing to schedule events and does not require any extra links in the table entries. We describe this alternative in the next section.

4. DESIGN AND ANALYSIS OF A HEAP-BASED CALLOUT TABLE

4.1 Definition and Description of a Heap

We define a *heap* as a complete binary tree in level order in which the key at each node in the tree is less than² or equal to the keys of its children^[4]. In the literature, heaps are also called priority queues. By this definition, a node by itself is a heap, every subtree of a heap is a heap, and the root of the tree has the smallest key value. An example of a heap is shown in Figure 1; the numbers in the nodes represent the values of the keys.

There are several convenient properties of heap data structures. No explicit links are required to represent the binary tree if the heap is stored as an array in a root first and left-to-right level order. In this scheme, the index in the array of the root is 0. For an arbitrary non-root node i , the parent of the node is $\lfloor (i-1)/2 \rfloor$, and the left and right children (if they exist) are $2i+1$ and $2i+2$ respectively. The array storage representation of the heap illustrated in Figure 1 is shown in Figure 2; the implicit links between parents and children are also sketched in.

Although a heap is not totally ordered, a significant amount of partial ordering information is embedded in the structure. Thus, we can expect the cost of maintaining the heap structure when nodes are inserted and deleted to be better than linear when this partial ordering information is used. For the operations required by the callout mechanism, a complete ordering is unnecessary and would be costly to maintain. Because the root of a heap is always the smallest entry in the tree, the entries percolate their way up to the root in a piece-meal fashion, conveniently arriving at the root when it is their turn to be processed

2. A heap can be defined with either the smallest element or the largest element on top. Furthermore, heaps are not necessarily binary trees, but for our problem we use only binary trees.

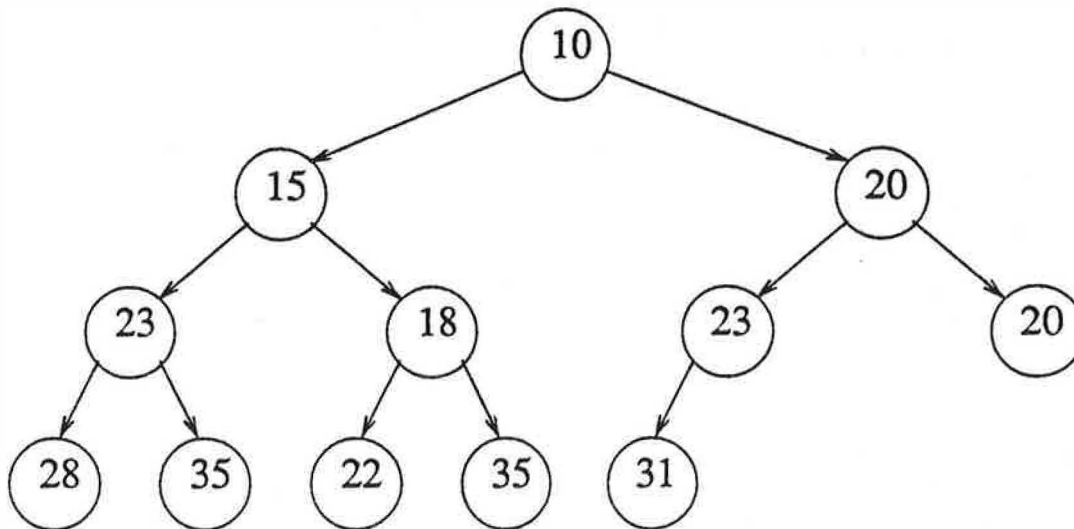


Figure 1. A Binary Heap

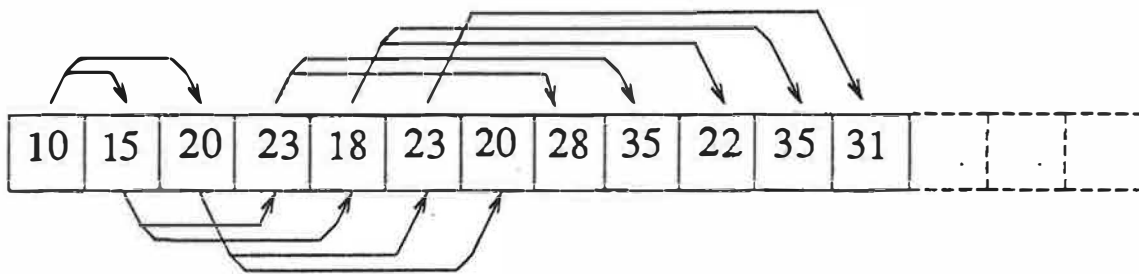


Figure 2. The Array Representation of a Binary Heap

next.

4.2 Schedule, De-Schedule, and Fire Operations

For a heap-based callout table, the *timeout()*, *untimeout()*, and *timein()* operations correspond to adding a new node to the heap, deleting an existing node from the heap, and deleting the root from the heap respectively. Implicit in the heap-based implementation is the requirement that we always maintain the heap structure; thus, the computational effort is reflected in the re-heaping we must do after each of these operations.

Re-heaping falls into two categories: heap-up and heap-down. If for some reason a node is out of place, we have to exchange it either with one of its two children (heap-down) or with its parent (heap-up). Specifically, if a node is not smaller than both of its children, we must exchange the node with the smaller of its two children. We once again compare the node, in its new position, with its new children, and repeat the heap-down process until the node settles in a position such that the heap is restored. If a node is smaller than its parent, we have to heap-up. We exchange the node with its parent and then re-examine it with respect to its parent in its new position; we continue heaping-up until the heap is restored. Because the heap is a complete binary tree, the depth of a n -node heap is $O(\log(n))$. Thus, both the heap-up and heap-down processes take $O(\log(n))$ time since they percolate along the tree structure towards the root or the leaves and touch at most $O(\log(n))$ nodes^[3]. With this framework, we now outline the work required for each of the three callout operations.

4.2.1 Scheduling An Event

Scheduling an event is a *timeout()* call that adds a node to the heap. This node is added to the first available position in the complete binary tree, i.e., at the end of its corresponding array representation. Since we had a legal heap before we added the node, this is the only node that may be out of place (the key value is, of course, its time-to-fire in the future). Because we place the new entry at the end of the array, there is no search phase involved; the re-organization phase is the heap-up process that takes $O(\log(n))$ time.

4.2.2 De-Scheduling An Event

De-scheduling an event is an *untimeout()* call that deletes a particular node from the heap. If we do not use any additional data structures, we must conduct a linear search for the specific node. Once we locate the node, we remove it and replace it with the last node of the heap (last element in the array) so that we can maintain the completeness of the binary tree. Depending on the relative value of this node with respect to its new parent and new children, we may have to heap-up or heap-down. If its key value is smaller than its parent, it will percolate its way up until the heap structure is restored. Otherwise, if it is larger than either of its children, it will have to percolate down until the heap structure is restored. The re-organization phase takes $O(\log(n))$ time.

4.2.3 Firing An Event

Firing an event is a *timein()* call that fires the event corresponding to the root node of the heap structure; this node is then removed. In terms of the re-organizational effort, deleting the root node is merely a special case of de-scheduling an event. The last node of the heap is placed at the root and the heap is restored by a heap-down process. There is no search phase, and the re-organization phase takes $O(\log(n))$ time.

4.3 Computational Complexity

Similar to Tables 1 and 2, in Table 3 we summarize the computational complexity of the three operations for a heap-based implementation.

TABLE 3. Computational Complexity in a Heap-Based Callout Table

Operation	Search	Re-organize
<i>Timeout()</i>	$O(1)$	$O(\log(n))$
<i>Untimeout()</i>	$O(n)$	$O(\log(n))$
<i>Timein()</i>	$O(1)$	$O(\log(n))$

Notice that the *untimeout()* operation is still dominated by the search phase. We will discuss a possible improvement in the next subsection.

A superficial comparison between the *timein()* costs of the heap-based and the linked list-based implementations may suggest that the heap-based implementation is not necessarily better. If, however, we consider the *amortized*³ cost^[5] of the *timein()* and *timeout()* pairs, we see that the heap-based implementation actually is more efficient in the amortized sense. By the simple measure itself, the *timein()* operation is indeed slower for the heap-based implementation. However, for each *timein()* operation, there has to have been an earlier corresponding *timeout()* operation. If we consider the combined cost for the *timeout()* and *timein()* pairs, the linked list version has $O(n)$ complexity while the heap version has only $O(\log(n))$ complexity. The analysis of the amortized cost gives a much more realistic comparison between algorithmic and data structure choices; the heap-based implementation is more efficient.

4.4 A Dictionary Based on Hashing for De-scheduling

We can improve the time to search for the event to delete when *untimeout()* is called by using hashing^[6]. We experimented with a hashing technique that masked off the high-order bits of the event

3. The amortized cost is basically the average cost over possible sequences of operations.

identifier and used the result as an index into hash buckets; collisions within a hash bucket were resolved via a linked list. Our measurements showed that the overhead of maintaining these hash chains was large, and this overhead occurs in all three operations since they all involve exchanges of nodes in the re-heaping process. Whenever we exchange two nodes, we must re-link the hash chains. Furthermore, the array representation of the heap makes the linear search for a match of a given identifier streamlined, and we decided not to use any dictionary to improve the search in this work.

Alternatively, we can reduce the search phase of the *untimeout()* operation to $O(1)$ by using a parallel *event identifier* (event id) structure. This structure is an array of pointers with as many entries as the callout table; initially, all event id entries are linked in a free list. When the system receives a *timeout()* request, it takes an event id entry from the free list. The *address* of the event id entry is used as the unique callout identifier, and the pointer to the corresponding callout table entry is kept in the event id entry. Whenever the callout entry is moved in reheap, the event id entry must be updated to point to its new location; the event identifier remains constant and does not need to be changed. Now the search for a callout event in an *untimeout()* request is merely an indirect reference through the event id structure; maintaining the pointers, however, adds to the costs for *timein()* and *timeout()*. When a callout entry is removed, either by *timein()* or *untimeout()*, the event id entry is linked back into the free list.

5. A PROTOTYPE BASED ON UNIX SYSTEM V RELEASE 2

To gather empirical performance data, we built a prototype of the heap-based callout table for a minicomputer running UNIX System V Release 2. We wanted to measure the cost of the operations in the new version and determine the number of entries in the table at which the heap-based callout table becomes more efficient than the standard implementation. (The callout table for System V Release 2 is array-based.) The important metrics are the average CPU times per call for *timeout()*, *untimeout()*, and *timein()* as a function of the number of entries in the table.

5.1 Kernel Changes

The only source file we changed was *os/clock.c*. We rewrote the *timeout()*, *untimeout()*, and *timein()* routines to use the heap-based callout table, and added two local support routines, *heap-up()* and *heap-down()*. The new implementation simplifies the way the clock interrupt decrements the time entry in the callout table, and also the way it checks for an event ready to be fired. Since the structure of the callout entry is identical to that of the array-based implementation, no changes were made to the *callo.h* header file. The heap-based implementation merely imposes and maintains the heap structure on the *same* array used in the original design. Furthermore, the interfaces to the callout routines are unchanged, and thus the heap-based version causes no compatibility problems. On a VAX-11/750[†], the text size of the array-based code for the callout mechanism is 480 bytes and the heap-based version is 664 bytes⁴. On an AT&T 3B2, the text size of the array-based code is 492 bytes and the size of the heap-based version is 872 bytes.

5.2 Measuring Call Cost Against Number of Entries in Table

To measure the performance of the three operations used in the callout mechanism, we designed a workload scenario that applied an easily controlled load to the underlying system. We create a pair of competing processes that go to sleep after scheduling a wakeup (via *timeout()*) some random time in the future. When the first of these two competing processes is awakened, it kills the other process by de-scheduling its wakeup (via *untimeout()*). Then the competition is started over again by drawing two more random numbers and scheduling two more wakeups. In this scenario, the wakeup corresponds to a *timein()* call. For each pair of competing processes we get 2 table entries; thus, by varying the number of pairs in an otherwise idle system, we can precisely control the number of entries in the callout table.

[†] Trademark of Digital Equipment Corporation.

4. With bucket hashing, the text size of the heap-based version increases to 1348 bytes.

The kernel was instrumented to trace each call by timestamping the entry to and exit from the *timeout()*, *untimeout()*, and *timein()* routines. The trace utility uses a low level hardware interval timer with 10 microsecond resolution for its timestamping^[6].

5.3 Analysis of Measurement Results

We analyzed the CPU times spent on calls to *timein()*, *timeout()*, and *untimeout()* routines by varying the number of pairs in the driving workload. The means and standard deviations for the three operations are tabulated in Tables 4, 5, and 6.

TABLE 4. Statistics for calls to *timein()*

	Array-based		Heap-based	
Complexity	$O(n)$		$O(\log(n))$	
Number of Pairs	Mean (msec)	Standard Deviation	Mean (msec)	Standard Deviation
2	0.259	0.005	0.228	0.020
4	0.353	0.006	0.307	0.027
8	0.541	0.001	0.370	0.041
16	0.920	0.041	0.434	0.041
32	1.687	0.103	0.501	0.045
64	3.206	0.096	0.567	0.084

TABLE 5. Statistics for calls to *timeout()*

	Array-based		Heap-based	
Complexity	$O(n)$		$O(\log(n))$	
Number of Pairs	Mean (msec)	Standard Deviation	Mean (msec)	Standard Deviation
2	0.160	0.011	0.183	0.021
4	0.189	0.017	0.189	0.041
8	0.249	0.030	0.194	0.056
16	0.371	0.066	0.196	0.064
32	0.611	0.117	0.197	0.072
64	1.058	0.227	0.192	0.081

TABLE 6. Statistics for calls to *untimeout()*

	Array-based		Heap-based	
Complexity	$O(n)$		$O(n)$	
Number of Pairs	Mean (msec)	Standard Deviation	Mean (msec)	Standard Deviation
2	0.169	0.009	0.189	0.025
4	0.229	0.033	0.204	0.060
8	0.350	0.078	0.261	0.053
16	0.580	0.176	0.315	0.048
32	1.052	0.342	0.392	0.067
64	1.984	0.678	0.529	0.111

Notice that for *timein()* the heap-based implementation is always better, and the pay back points for the *timeout()* and *untimeout()* operations are less than 4 entries. If we consider the pairs of *timeout()* and *timein()* calls, the amortized cost of the heap-based version is less than that of the standard array-based version even with only 2 entries in the table. Only *untimeout()* is initially slightly more expensive. These "cross-over" points are low in spite of the added overhead implicit in maintaining the heap structure; the approach obtains dramatic CPU cost improvements at the expense of a small increase in

code size. Furthermore, the standard deviation for the heap-based implementation is much smaller than the array-based version when the table size is large. This characteristic is required for real-time applications.

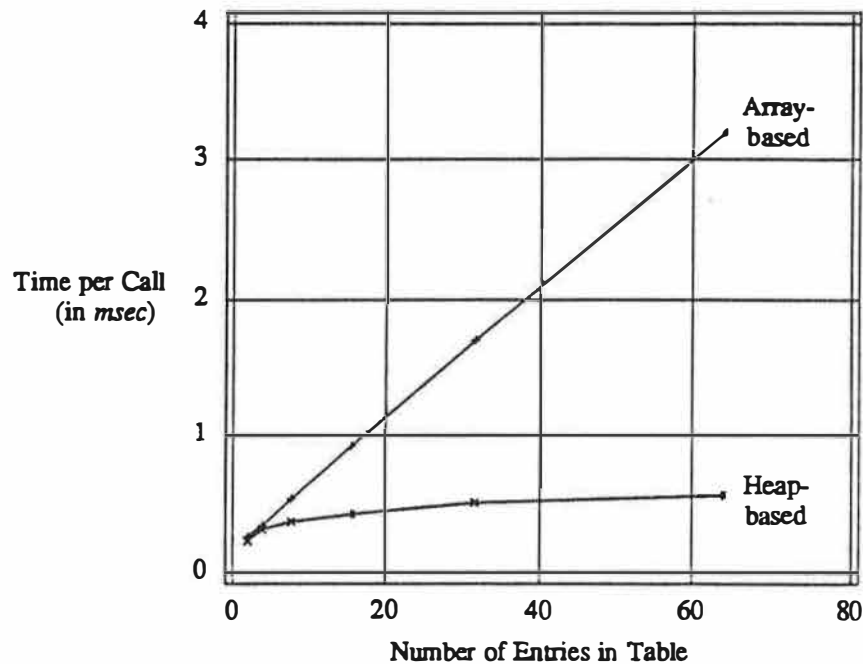


Figure 3. Average Time for `timein()`

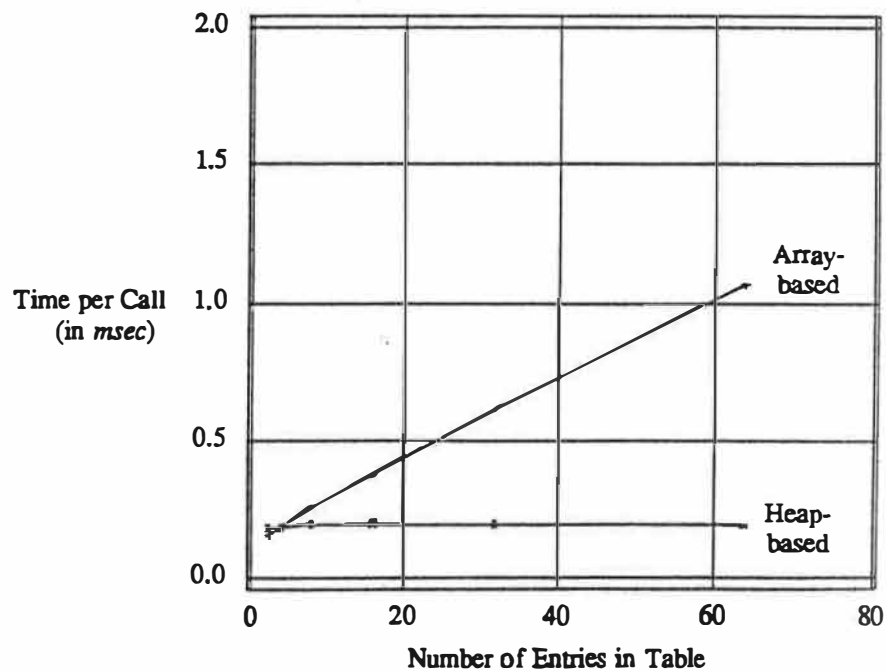


Figure 4. Average Time for `timeout()`

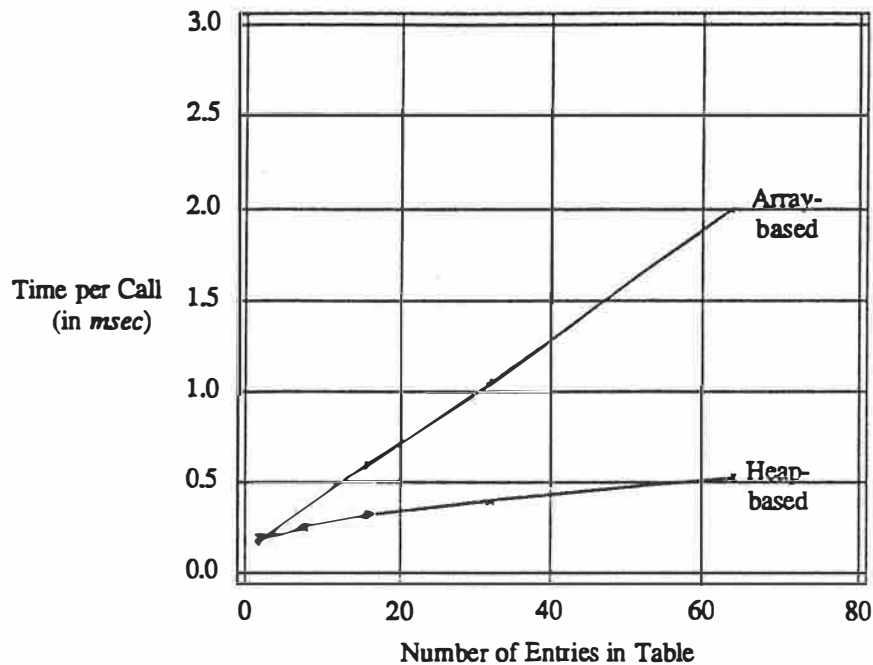


Figure 5. Average Time for *untimeout()*

The mean times per call as a function of load for these three operations are graphed in Figures 3, 4, and 5. The logarithmic CPU cost for the heap-based version and the linear cost of the array-based version are evident from the graphs. It is also interesting to observe in Figure 5 the (small) linear component of *untimeout()* for the heap-based implementation.

5.4 The Delayed Event Heuristic for Further Improvement

To further reduce the costs of *timeout()*, we can use a *delayed event* heuristic. Based on ideas used in multiway merging and replacement selection^[7], this heuristic buffers a pending event, cutting the average cost of *timeout()* in half. The heuristic works as follows: we reserve an entry after the end of array (storage) representation of the heap structure for the delayed event; a binary variable shows whether a delayed event exists in the reserved slot. Whenever a *timeout()* call takes place, if there is no delayed event, this new event is the delayed event. If a delayed event already exists, the delayed event is placed in the heap by the heap-up process, and the new event becomes the delayed event. If the new delayed event has a smaller firing time than that of the root, we exchange it with the root entry. When a *timein()* or *timeout()* occurs, we use the delayed event to fill the newly vacated entry and heap up or down as appropriate. This heuristic works best when *timein()* (or *untimeout()*) alternates with the *timeout()* calls; in that case the cost of *timeout()* practically disappears since we have only the function call, several comparisons, and a structure copy.

Since such alternating events are unlikely to occur in practice, we can predict the expected savings by the following analysis. Under the operational constraints, the number of *timeout()* calls is equal to the number of *timein()* and *untimeout()* calls over the long run. In other words, the arrival rate (λ) to the callout table is the same as the departure rate from the table. Let us define state 0 as the state of having no delayed event and state 1 as the state of having a delayed event. By assuming the times between callout events are distributed exponentially, we can derive the saving by using a simple two-state continuous-time Markov chain^[8] shown in Figure 6.

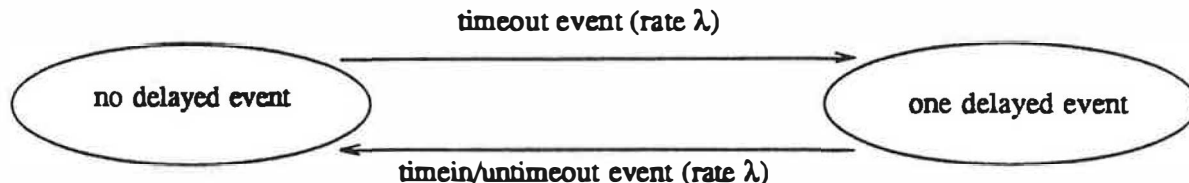


Figure 6. A Two-state Continuous Time Markov Chain

The 2x2 matrix of transition rates Q is as follows:

$$Q = \begin{bmatrix} -\lambda & \lambda \\ \lambda & -\lambda \end{bmatrix}$$

and the stationary state probability $\Pi = (\pi_0, \pi_1) = (1/2, 1/2)$ can be readily solved using $\Pi Q = 0$ subject to constraints $\pi_i \geq 0$ for $i \in \{0, 1\}$ and $\sum_{i=0}^1 \pi_i = 1$. Thus, the fraction of time we have one delayed event is the same as that in which we have no delayed event. Since we have 0.50 probability of using the delayed event if the next call is a departure (`timein()` or `untimeout()`), half of the timeout events will never go through the heap up process. A small amount of extra code in the callout implementation shows this saving without noticeably changing the other costs.

6. SUMMARY

We have described the design and analysis of an alternative implementation of the callout table based on a heap structure, and have reported the results of various experiments conducted on a prototype of the design. Measurements show that a heap-based callout table is much more efficient and predictable than the standard array-based table, and is thus better suited for the needs of real-time applications. We also discuss using absolute timing in the callout mechanism; this idea better captures the essence of event ordering and allows implementations that use different data structures. Finally, we described a delayed event heuristic that buffers events without re-heaping, and this cuts the average `timeout()` cost in half.

REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal*, Vol. 57, No.6, Part 2, July-August 1978.
2. M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Company, Reading, Massachusetts, 1974.
4. T. A. Standish, *Data Structure Techniques*, Addison-Wesley Company, Reading, Massachusetts, 1980.
5. R. E. Tarjan, "Amortized Computational Complexity," *SIAM Journal on Algebraic and Discrete Methods*, Vol. 6 No.2, April 1985.
6. R. E. Barkley and D. Chen, "CASPER the Friendly Daemon," paper in preparation.
7. D. E. Knuth, *The Art of Computer Programming, Vol III, Sorting and Searching*, Addison-Wesley, Reading, Mass. 1972.
8. D. P. Heyman and M. J. Sobel, *Stochastic Models in Operations Research, Volume I, Stochastic Processes and Operating Characteristics*, McGraw-Hill, New York, 1982.

A Memory Allocation Profiler for C and Lisp Programs

Benjamin Zorn

Paul Hilfinger

Computer Science Division, Dept. of EECS
University of California at Berkeley

Abstract

This paper describes `mprof`, a tool used to study the dynamic memory allocation behavior of programs. `Mprof` records the amount of memory that a function allocates, breaks down allocation information by type and size, and displays a program's dynamic call graph so that functions indirectly responsible for memory allocation are easy to identify. `Mprof` is a two-phase tool. The monitor phase is linked into executing programs and records information each time memory is allocated. The display phase reduces the data generated by the monitor and presents the information to a user in several tables. `Mprof` has been implemented for C and Kyoto Common Lisp. Measurements of these implementations are presented.

1 Introduction

Dynamic memory allocation, hereafter referred to simply as memory allocation, is an important part of many programs. By dynamic allocation, we mean the memory allocated from the heap. Unnecessary allocation can decrease program locality and increase execution time for the allocation itself and for possible memory reclamation. If reclamation is not performed, or if some objects are accidentally not reclaimed (a "memory leak"), programs can fail when they reach the memory size limit. Programmers often write their own versions of memory allocation routines to measure and reduce allocation overhead. It is estimated that Mesa programmers spent 40% of their time solving storage-management related problems before automatic storage reclamation techniques were introduced in Cedar [7]. Even with automatic storage management, in which reclamation occurs transparently, memory allocation has a strong effect on the performance of programs [4]. Although memory allocation is important, few software tools exist to help programmers understand the memory allocation behavior of their programs.

`Mprof` is a tool that allows programmers to identify where and why dynamic memory is allocated in a program. It records which functions are directly responsible for memory allocation and also records the dynamic call chain at an allocation to show which functions were indirectly responsible for the allocation. The design of `mprof` was inspired by the tool

This research was funded by DARPA contract number N00039-85-C-0269 as part of the SPUR research project.

`gprof`, a dynamic execution profiler [2]. `gprof` is a very useful tool for understanding the execution behavior of programs; `mprof` extends the ideas of `gprof` to give programmers information about the dynamic memory allocation behavior of their programs. `Mprof` is a two-phase tool. A monitor phase is linked into an executing application and collects data as the application executes. A reduction and display phase takes the data collected by the monitor and presents it to the user in concise tables.

A profiling program such as `mprof` should satisfy several criteria. First, a program monitor should not significantly alter the behavior of the program being monitored. In particular, a monitor should not impose so much overhead on the program being monitored that large programs cannot be profiled. Second, a monitor should be easy to integrate into existing applications. To use `mprof`, programmers simply have to relink their applications with a special version of the system library. No source code modifications are required. Finally, a monitor should provide a programmer with information that he can understand and use to reduce the memory allocation overhead of his programs. We will present an example that illustrates such a use of `mprof`.

In Section 2, we present a simple program and describe the use of `mprof` with respect to the example. In Section 3 we discuss techniques for the effective implementation of `mprof`. Section 4 presents some measurements of `mprof`. Section 5 describes other memory profiling tools and previous work on which `mprof` is based, while Section 6 contains our conclusions.

2 Using `mprof`

2.1 A Producer/Consumer Example

To illustrate how `mprof` helps a programmer understand the memory allocation in his program, consider the C program in Figure 1. In this program, a simplified producer/consumer simulation, objects are randomly allocated by two producers and freed by the consumer. The function `random_flip`, which is not shown, randomly returns 1 or 0 with equal probability. The function `consume_widget`, which is responsible for freeing the memory allocated, contains a bug and does not free red widgets. If the simulation ran for a long time, memory would eventually be exhausted, and the program would fail.

In the example, `make_widget` is the only function that allocates memory directly. To fully understand the allocation behavior of the program, we must know which functions called `make_widget` and hence were indirectly responsible for memory allocation. `Mprof` provides this information.

To use `mprof`, programmers link in special versions of the system functions `malloc` and `free`, which are called each time memory is allocated and freed, respectively. The application is then run normally. The `mprof` monitor function, linked in with `malloc`, gathers statistics as the program runs and writes this information to a file when the application exits. The programmer then runs a display program over the data file, and four tables are printed: a list of memory leaks, an allocation bin table, a direct allocation table, and a dynamic call graph.

```

typedef struct {
    enum color c;
    int data[50];
} widget;

#define WSIZE sizeof(widget)

widget *
make_widget()
{
    widget    *w;

    w = (widget *) malloc(WSIZE);
    return w;
}

widget *
make_blue_widget()
{
    widget    *w;

    w = make_widget();
    w->c = BLUE;
    return w;
}

widget *
make_red_widget()
{
    widget    *w;

    w = make_widget();
    w->c = RED;
    return w;
}

void
consume_widget(w)
widget *w;
{
    if (w->c == BLUE) {
        /* record blue widget */
        free(w);
    } else {
        /* record red widget */
    }
}

#define NUM_WIDGETS    10000

int
main()
{
    int        i;
    widget    *wqueue[NUM_WIDGETS];

    for (i = 0; i < NUM_WIDGETS; i++)
        if (random_flip())
            wqueue[i] = make_blue_widget();
        else
            wqueue[i] = make_red_widget();

    for (i = 0; i < NUM_WIDGETS; i++)
        consume_widget(wqueue[i]);

    return 0;
}

```

Figure 1: A Simple Producer/Consumer Simulation Program

Each table presents the allocation behavior of the program from a different perspective. The rest of this section describes each of the tables for the C program in Figure 1.

2.2 The Memory Leak Table

C programmers must explicitly free memory objects when they are done using them. Memory leaks arise when programmers accidentally forget to release memory. Because Lisp reclaims memory automatically, the memory leak table is not necessary in the Lisp version of mprof.

The memory leak table tells a programmer which functions allocated the memory associated with memory leaks. The table contains a list of partial call paths that resulted in memory that was allocated and not subsequently freed. The paths are partial because complete path information is not recorded; only the last five callers on the call stack are listed in the memory leak table. In our simple example, there is only one such path, and it tells us immediately what objects are not freed. Figure 2 contains the memory leak table for our example.

allocs	bytes (%)	path
5019	1023876 (99)	> main > make_red_widget > make_widget

Figure 2: Memory Leak Table for Producer/Consumer Example

In larger examples, more than one path through a particular function is possible. We provide an option that distinguishes individual call sites within the same function in the memory leak table if such a distinction is needed.

2.3 The Allocation Bin Table

A major part of understanding the memory allocation behavior of a program is knowing what objects were allocated. In C, memory allocation is done by object size; the type of object being allocated is not known at allocation time. The allocation bin table provides information about what sizes of objects were allocated and what program types correspond to the sizes listed. This knowledge helps a programmer recognize which data structures consume the most memory and allows him to concentrate any space optimizations on them.

The allocation bin table breaks down object allocation by the size, in bytes, of allocated objects. Figure 3 shows the allocation bin table for the program in Figure 1.

The allocation bin table contains information about objects of each byte size from 0 to 1024 bytes and groups objects larger than 1024 bytes into a single bin. For each byte size in which memory was allocated, the allocation bin table shows the number of allocations

size:	allocs	bytes (%)	frees	kept (%)	types
204	10000	2040000 (99)	4981	1023876 (99)	widget
> 1024	0	0	0	0	
<TOTAL>	10000	2040000	4981	1023876	

Figure 3: Allocation Bin Table for Producer/Consumer Example

of that size (**allocs**), the total number of bytes allocated to objects of that size (**bytes**), the number of frees of objects of that size (**frees**), the number of bytes not freed that were allocated to objects of that size (**kept**¹), and user types whose size is the same as the bin size (**types**). From the example, we can see that 10,000 widgets were allocated by the program, but only 4,981 of these were eventually freed, resulting in 1,023,876 bytes of memory lost to the memory leak. The percentages show what fraction of all bins a particular bin contributed. This information is provided to allow a user to rapidly determine which bins are of interest (i.e., contribute a substantial percentage). 99% is the largest percentage possible because we chose to use a 2 character field width.

2.4 The Direct Allocation Table

Another facet of understanding memory allocation is knowing which functions allocated memory and how much they allocated. In C, memory allocation is performed explicitly by calling `malloc`, and so programmers are often aware of the functions that allocate memory. Even in C, however, knowing how much memory was allocated can point out functions that do unnecessary allocation and guide the programmer when he attempts to optimize the space consumption of his program. In Lisp, memory allocation happens implicitly in many primitive routines such as `mapcar`, `*`, and `intern`. The direct allocation table can reveal unsuspected sources of allocation to Lisp programmers.

Figure 4 contains the direct allocation table for our example. The direct allocation table corresponds to the flat profile generated by `gprof`.

The first line of the direct allocation table contains the totals for all functions allocating memory. In this example, only one function, `make_widget`, allocates memory. The direct allocation table prints percent of total allocation that took place in each function (**% mem**), the number of bytes allocated by each function (**bytes**), the number of bytes allocated by the function and never freed (**bytes kept**), and the number of calls made to the function that resulted in allocation (**calls**). The **% mem(size)** fields contain a size breakdown²

¹The label **kept** is used throughout the paper to refer to objects that were never freed.

²Both the direct allocation table and the dynamic call graph break down object allocation into four

% mem	bytes	% mem(size)	bytes kept	% all kept	calls	name
		s m l x		s m l x		
—	2040000	99	1023876	99	10000	<TOTAL>
100.0	2040000	99	1023876	99	10000	make_widget

Figure 4: Direct Allocation Table for Producer/Consumer Example

of the memory allocated by each function as a fraction of the memory allocated by all functions. In this example, 99% of the memory allocated by the program was allocated in `make_widget` for medium-sized objects. Blank columns indicate values less than 1%. The other size breakdown given in the direct allocation table is for the memory that was allocated and never freed. The % all kept field contains a size breakdown of the memory not freed by a particular function as a fraction of all the memory not freed. In the example, 99% of the unfreed memory was of medium-sized objects allocated by `make_widget`.

2.5 The Allocation Call Graph

Understanding the memory allocation behavior of a programs sometimes requires more information than just knowing the functions that are directly responsible for memory allocation. Sometimes, as happens in Figure 1, the same allocation function is called by several different functions for different purposes. The allocation call graph shows all the functions that were indirect callers of functions that allocated memory.

Because the dynamic caller/callee relations of a program are numerous, the dynamic call graph is a complex table with many entries. Often, the information provided by the first three tables is enough to allow programmers to understand the memory allocation of their program. Nevertheless, for a full understanding of the allocation behavior of programs the allocation call graph is useful. Figure 5 contains the allocation call graph for the producer/consumer example and corresponds to the call graph profile generated by `gprof`.

The allocation call graph is a large table with an entry for each function that was on a call chain when memory was allocated. Each table entry is divided into three parts. The line for the function itself (called the *entry function*); lines above that line, each of which represents a caller of the entry function (the ancestors), and lines below that line, each of which represents a function called by the entry function (the descendents). The entry function is easy to identify in each table entry because a large rule appears in the `frac` column on that row. In the first entry of Figure 5, `main` is the entry function; there are no ancestors and two descendents.

categories of object size: small (s), from 0–32 bytes; medium (m), from 33–256 bytes; large (l), from 257–2048 bytes; and extra large (x), larger than 2048 bytes. For Lisp, categorisation is by type rather than size: cons cell (c), floating point number (f), structure or vector (s), and other (o).

	self	/ances	/ances	called/total	ancestors
index + desc	self (%)	size-func \desc	frac \desc	called/recur called/total	[name [index] descendents]
<hr/>					
[0]	100.0	0 (0)		0	main [0]
		1023876 (50)	99	50	make_red_widget [2]
		1016124 (49)	99	49	make_blue_widget [3]
	all	2040000	99	4981/4981	
<hr/>					
	all	2040000	99		
		1023876 (50)	99	50	make_red_widget [2]
		1016124 (49)	99	49	make_blue_widget [3]
[1]	100.0	2040000 (100)	99	10000	make_widget [1]
<hr/>					
[2]	50.2	1023876 (100)	99	99	main [0]
		0 (0)			make_red_widget [2]
		1023876 (100)	99	99	make_widget [1]
<hr/>					
[3]	49.8	1016124 (100)	99	99	main [0]
		0 (0)			make_blue_widget [3]
		1016124 (100)	99	99	make_widget [1]
<hr/>					

Figure 5: Allocation Call Graph for Producer/Consumer Example

The entry function line of the allocation call graph contains information about the function itself. The `index` field provides a unique index to help users navigate through the call graph. The `self + desc` field contains the percent of total memory allocated that was allocated in this function and its descendents. The call graph is sorted by decreasing values in this field. The `self` field contains the number of bytes that were allocated directly in the entry function. The `size-func` fields contain a size breakdown of the memory allocated in the function itself. Some functions, like `main` (index 0) allocated no memory directly, so the `size-func` fields are all blank. The `called` field shows the number of times this function was called during a memory allocation, with the number of recursive calls recorded in the adjacent field.

Each caller of the entry function is listed on a separate line above it. A summary of all callers is given on the top line of the entry if there is more than one ancestor. The `self` field of `ancestors` lists the number of bytes that the entry function and its descendents allocated on behalf of the ancestor. The `size-ances` field breaks down those bytes into size categories, while the `frac-ances` field shows the size breakdown of the bytes requested by this ancestor as a fraction of bytes allocated at the request of all ancestors. For example, in the entry for function `make_widget` (index 1), the ancestor `make_red_widget` can be

seen to have requested 1,023,876 bytes of data from `make_widget`, 99% of which was of medium-sized objects. Furthermore, calls from `make_red_widget` accounted for 50% of the total memory allocated by `make_widget` and its descendents. Other fields show how many calls the ancestor made to the entry function and how many calls the ancestor made in total. In a similar fashion, information about the function's descendents appears below the entry function.

Had the memory leak table not already told us what objects were not being freed, we could use the allocation call graph for the same purpose. The direct allocation table told us that `make_widget` allocated 1,023,876 bytes of unfreed memory, all for medium-sized objects. From the allocation call graph, we can see that the function `make_red_widget` was the function calling `make_widget` that requested 1,023,876 bytes of medium-sized objects.

Cycles in the call graph are not illustrated in Figure 5. As described in the next section, cycles obscure allocation information among functions that are members of a cycle. When the parent/child relationships that appear in the graph are between members of the same cycle, most of the fields in the graph must be omitted.

3 Implementation

We have implemented `mprof` for use with C and Common Lisp programs. Since the implementations are quite similar, the C implementation will be described in detail, and the minor differences in the Lisp implementation will be noted at the end of the section.

3.1 The Monitor

The first phase of `mprof` is a monitor that is linked into the executing application. The monitor includes modified versions of `malloc` and `free` that record information each time they are invoked. Along with `malloc` and `free`, `mprof` provides its own `exit` function, so that when the application program exits, the data collected by the monitor is written to a file. The monitor maintains several data structures needed to construct the tables.

To construct the leak table, the monitor associates a list of the last five callers in the call chain, the *partial call chain*, with the object allocated. `mprof` augments every object allocated with two items: an integer which is the object size as requested by the user (since the allocator may allocate an object of a different size for convenience), and a pointer to a structure that contains the object's partial call chain and a count of allocations and frees of objects with that call chain. A hash table is used to map a partial call chain to the structure containing the counters. When an object is allocated, its partial call chain is used as a hash key to retrieve the structure containing the counters. A pointer to the structure is placed in the allocated object and the allocation counter is incremented. When the object is later freed, the pointer is followed and the counter of frees is incremented. Any partial call chain in which the number of allocations does not match the number of frees indicates a memory leak and is printed in the leak table.

To construct the allocation bin table, the monitor has a 1026-element array of integers to count allocations and another 1026-element array to count frees. When objects of a particular size from 0–1024 bytes are allocated or freed, the appropriated bin is incremented. Objects larger than 1024 bytes are grouped into the same bin.

The construction of the direct allocation table falls out directly from maintaining the allocation call graph information, which is described in the next section.

3.2 Constructing the Allocation Call Graph

To construct the allocation call graph, the monitor must associate the number of bytes allocated with every function on the current dynamic call chain, each time `malloc` is called. Consider the sample call chain in Figure 6, which we abbreviate: `main->foo->bar(24)`.

CALL STACK:	MPROF RECORDS:
main calls foo	24 bytes over main -> foo
foo calls bar	24 bytes over foo -> bar
bar calls malloc(24)	24 bytes allocated in bar

Figure 6: Example of a Dynamic Call Chain

In `mprof`, the monitor traverses the entire call chain by following return addresses. This differs from `gprof`, where only the immediate caller of the current function is recorded. `gprof` makes the assumption that each call takes an equal amount of time and uses this assumption to reconstruct the complete dynamic call graph from information only about the immediate callers. In `mprof`, we actually traverse the entire dynamic call chain and need to make no assumptions.

In choosing to traverse the entire call chain, we have elected to perform an operation that is potentially expensive both in time and space. One implementation would simply record every function in every chain and write the information to a file (i.e., in the example we would output `[main->foo->bar, 24]`). Considering that many programs execute millions of calls to `malloc` and that the depth of a call chain can be hundreds of functions, the amount of information could be prohibitive.

An alternative to recording the entire chain of callers is to break the call chain into a set of caller/callee pairs, and associate the bytes allocated with each pair in the chain. For the call in the example, we could maintain the pairs `[main, foo]` and `[foo, bar]`, and associate 24 bytes with each pair. Conceptually, the data structure our monitor maintains is an association between caller/callee pairs and the cumulative bytes allocated over the pair, which we denote `([main, foo], 24)`. To continue with the example, if the next allocation was: `main->foo->otherbar(10)`, where this is the first call to `otherbar`, we would update the byte count associated with the `[main, foo]` pair to 34 from 24. Furthermore, we would create a new association between `[foo, otherbar]` and the byte count, 10. A disadvantage with this implementation is that the exact call chains are no longer available. However,

from the pairs we can construct the correct dynamic call graph of the program, which is the information that we need for the allocation call graph.

For the overhead imposed by the monitor to be reasonable, we have to make the association between caller/callee pairs and cumulative byte counts fast. We use a hash table in which the hash function is a simple byte-swap XOR of the callee address. Each callee has a list of its callers and the number of allocated bytes associated with each pair. In an effort to decrease the number of hash lookups, we noted that from allocation to allocation, most of the call chain remains the same. Our measurements show that on the average, 60–75% of the call chain remains the same between allocations. This observation allows us to cache the pairs associated with the current caller chain and to use most of these pairs the next time a caller chain is recorded. Thus, on any particular allocation, only a few addresses need to be hashed. Here are the events that take place when a call to `malloc` is monitored:

1. The chain of return addresses is stored in a vector.
2. The new chain is compared with the previous chain, and the point at which they differ is noted.
3. For the addresses in the chain that have not changed, the caller/callee byte count for each pair is already available and is incremented.
4. For new addresses in the chain, each caller/callee byte count is looked up and updated.
5. For the tail of the chain (i.e., the function that called `malloc` directly), the direct allocation information is recorded.

Maintaining allocation call graph information requires a byte count for every distinct caller/callee pair in every call chain that allocates memory. Our experience is that there are a limited number of such pairs, even in very large C programs, so that the memory requirements of the `mprof` monitor are not large (see section 4.2).

3.3 Reduction and Display

The second phase of `mprof` reads the output of the monitor, reduces the data to create a dynamic call graph, and displays the data in four tables. The first part of the data reduction is to map the caller/callee address pairs to actual function names. A program `mpfilt` reads the executable file that created the monitor trace (compiled so that symbol table information is retained), and outputs a new set of function caller/callee relations. These relations are then used to construct the subset of the program's dynamic call graph that involved memory allocation.

The call graph initially can contain cycles due to recursion in the program's execution. Cycles in the call graph introduce spurious allocation relations, as is illustrated in Figure 7. In this example, `main` is credited as being indirectly responsible for 10 bytes, but because we only keep track of caller/callee pairs, `F` appears to have requested 20 bytes from `G`, even though only 10 bytes were allocated.

CALL STACK:	MPROF RECORDS:
main calls F	(10 bytes over main -> F)
F calls G	(10 bytes over F -> G)
G calls F	(10 bytes over G -> F)
F calls G	(10 MORE bytes over F -> G)
G calls malloc(10)	(10 bytes allocated in G)

Figure 7: Problems Caused by Recursive Calls

We considered several solutions to the problems caused by cycles and adopted the most conservative solution. One way to avoid recording spurious allocation caused by recursion is for the monitor to identify the cycles before recording the allocation. For example, in Figure 7, the monitor could realize that it had already credited F with the 10 bytes when it encountered F calling G the second time. This solution adds overhead to the monitor and conflicts with our goal to make the monitor as unobtrusive as possible.

The solution that we adopted was to merge functions that are in a cycle into a single node in the reduction phase. Thus, each strongly connected component in the dynamic call graph is merged into a single node. The result is a call graph with no cycles. This process is also used by gprof, and described carefully elsewhere [2]. Such an approach works well in gprof because C programs, for which gprof was primarily intended, tend to have limited amounts of recursion. Lisp programs, for which mprof is also intended, intuitively contain much more recursion. We have experience profiling a number of large Common Lisp programs. We observe several recursive cycles in most programs, but the cycles generally contain a small percentage of the total functions and mprof is quite effective.

3.4 Lisp Implementation

So far, we have described the implementation of mprof for C. The Lisp implementation is quite similar, and here we describe the major differences. C has a single function, `malloc`, that is called to allocate memory explicitly. Lisp has a large number of primitives that allocate memory implicitly (i.e., `cons`, `*`, `intern`, etc.). To make mprof work, these primitives must be modified so that every allocation is recorded. Fortunately, at the Lisp implementation level, all memory allocations may be channeled through a single routine. We worked with KCL (Kyoto Common Lisp), which is implemented in C. In KCL, all Lisp memory allocations are handled by a single function, `alloc_object`. Just as we had modified `malloc` in C, we were able to simply patch `alloc_object` to monitor memory allocation in KCL.

The other major difference in monitoring Lisp is that the addresses recorded by the monitor must be translated into Lisp function names. Again, KCL makes this quite easy because Lisp functions are defined in a central place in KCL and the names of the functions are known when they are defined. Many other Lisp systems are designed to allow return addresses to be mapped to symbolic function names so that the call stack can be printed

at a breakpoint. In this case, the monitor can use the same mechanism to map return addresses to function names. Therefore, in Lisp systems in which addresses can be quickly mapped to function names, memory profiling in the style of `mprof` is not a difficult problem. In systems in which symbolic names are not available in compiled code, profiling is more difficult. Furthermore, many systems open-code important allocation functions, like `cons`. Because open-coded allocation functions will not necessarily call a central allocation function (like `alloc_object`), such allocations will not be observed by `mprof`. To avoid such a loss of information, `mprof` should be used in conjunction with program declarations that will force allocation functions such as `cons` to be coded out-of-line.

4 Measurements

We have measured the C implementation of `mprof` by instrumenting four programs using `mprof`. The first program, `example`, is our example program with the number of widgets allocated increased to 100,000 to increase program execution time. The second program, `fidilrt`, is the runtime library of FIDIL, a programming language for finite difference computations [3]. The third program, `epoxy`, is an electrical and physical layout optimizer written by Fred Obermeier [5]. The fourth program, `crystal`, is a VLSI timing analysis program [6]. These tests represent a small program (`example`, 100 lines); a medium-sized program (`fidilrt`, 7,100 lines); and two large programs (`epoxy`, 11,000 lines and `crystal`, 10,500 lines). In the remainder of this section, we will look at the resource consumption of `mprof` from two perspectives: execution time overhead and space consumption.

4.1 Execution Time Overhead

There are two sources of execution time overhead associated with `mprof`: additional time spent monitoring an application and the time to reduce and print the data produced by the monitor. The largest source of monitor overhead is the time required to traverse the complete call chain and associate allocations with caller/callee pairs. We implemented a version of `mprof`, called `mprof-`, which does not create the allocation call graph. With this version, we can see the relative cost of the allocation call graph. The ratio of the time spent with profiling to the time spent without profiling is called the *slowdown factor*. Table 1 summarizes the execution time overheads for our four applications. Measurements were gathered running the test programs on a VAX 8800 with 80 megabytes of physical memory.

The slowdown associated with `mprof` varies widely, from 1.5 to 10. `crystal` suffered the worst degradation from profiling because `crystal` uses a depth-first algorithm that results in long call chains. Programs without long call chains appear to slow down by a factor of 2–4. If the allocation call graph is not generated and long call chains are not traversed, the slowdown is significantly less, especially in the extreme cases. Since `mprof` is a prototype and has not been carefully optimized, this overhead seems acceptable. From the table, we see the reduction and display time is typically less than a minute.

Resource Description	Cost			
	example	fidilrt	epoxy	crystal
Number of allocations	100000	77376	306295	31158
Execution time with <code>mprof</code> (seconds)	62.7	132.7	188.8	134.1
Execution time with <code>mprof-</code> (seconds)	44.1	116.0	149.7	25.5
Execution time without <code>mprof</code> (seconds)	17.9	107.1	52.1	13.2
Slowdown using <code>mprof</code>	3.5	1.2	3.6	10.1
Slowdown using <code>mprof-</code>	2.5	1.1	2.9	1.9
Reduction and display time (seconds)	10.3	28.8	28.3	36.8

Table 1: Execution Time Overhead of `mprof`

4.2 Storage Consumption

The storage consumption of `mprof` is divided into the additional memory needed by the monitor as an application executes, and the external storage required by the profile data file. The most significant source of memory used by the monitor is the data stored with each object allocated: an object size and a pointer needed to construct the memory leak table. The monitor also uses memory to record the memory bins and caller/callee byte counts and must write this information to a file when the application is finished. We measured how many bytes of memory and disk space are needed to store this information. Table 2 summarizes the measurements of storage consumption associated with `mprof`.

Resource Description	Cost			
	example	fidilrt	epoxy	crystal
Number of allocations	100000	61163	306295	31158
User memory allocated (Kbytes)	20000	2425	6418	21464
Per object memory (Kbytes)	781	477	2393	168
Other monitor memory (Kbytes)	8.7	23.3	52.3	17.5
Total monitor memory (Kbytes)	790	500	2445	186
Monitor fraction (% memory allocated)	4	17	28	1
Data file size (Kbytes)	4.5	8.1	28.6	9.6

Table 2: Storage Consumption of `mprof`

The memory overhead of `mprof` is small, except that an additional 8 bytes of storage are allocated with every object. In programs in which many small objects are allocated, like `epoxy`, `mprof` can contribute significantly to the total memory allocated. Nevertheless, in the worst case, `mprof` increases application size by 1/3, and since `mprof` is a development tool, this overhead seems acceptable. From the table we also see that the data files created by `mprof` are quite small (< 30 Kbytes).

5 Related Work

Mprof is similar to the tool gprof [2], a dynamic execution profiler. Because some of the problems of interpreting the dynamic call graph are the same, we have borrowed these ideas from gprof. Also, we have used ideas from the user interface of gprof for two reasons: because the information being displayed is quite similar and because users familiar with gprof would probably also be interested in mprof and would benefit from a similar presentation.

Barach, Taenzer, and Wells developed a tool for finding storage allocation errors in C programs [1]. Their approach concentrated on finding two specific storage allocation errors: memory leaks and duplicate frees. They modified malloc and free so that every time that these functions were called, information about the memory block being manipulated was recorded in a file. A program that examines this file, prleak, prints out which memory blocks were never freed or were freed twice. This approach differs from mprof in two ways. First, mprof provides more information about the memory allocation of programs than prleak, which just reports on storage errors. Second, prleak generates extremely large intermediate files that are comparable in size to the total amount of memory allocated by the program (often megabytes of data). Although mprof records more useful information, the data files it generates are of modest size (see the table above).

6 Conclusions

We have implemented a memory allocation profiling program for both C and Common Lisp. Our example has shown that mprof can be effective in elucidating the allocation behavior of a program so that a programmer can detect memory leaks and identify major sources of allocation.

Unlike gprof, mprof records every caller in the call chain every time an object is allocated. The overhead for this recording is large but not impractically so, because we take advantage of the fact that a call chain changes little between allocations. Moreover, recording this information does not require large amounts of memory because there are relatively few unique caller/callee address pairs on call chains in which allocation takes place, even in very large programs. We have measured the overhead of mprof, and find that typically it slows applications by a factor of 2–4 times, and adds up to 33% to the memory allocated by the application. Because mprof is intended as a development tool, these costs are acceptable.

Because mprof merges cycles caused by recursive function calls, mprof may be ineffective for programs with large cycles in their call graph. Only with more data will we be able to decide if many programs (especially those written in Lisp) contain so many recursive calls that cycle merging makes mprof ineffective. Nevertheless, mprof has already been effective in detecting KCL system functions that allocate memory extraneously.³

³Using mprof, we noted that for a large object-oriented program written in KCL, the system function every accounted for 13% of the memory allocated. We rewrote every so it would not allocate any memory,

As a final note, we have received feedback from C application programmers who have used mprof. They report that the memory leak table and the allocation bin table are both extremely useful, while the direct allocation table and the allocation call graph are harder to understand and also less useful. Considering the execution overhead associated with the allocation call graph and the complexity of the table, it is questionable whether the allocation call graph will ever be as helpful C programmers as the memory leak table. On the other hand, with automatic storage reclamation, the memory leak table becomes unnecessary. Yet for memory intensive languages, such as Lisp, the need for effective use of the memory is more important, and tools such as the allocation call graph might prove very useful. Because we have limited feedback from Lisp programmers using mprof, we cannot report their response to this tool.

References

- [1] David R. Barach, David H. Taenzer, and Robert E. Wells. A technique for finding storage allocation errors in C-language programs. *ACM SIGPLAN Notices*, 17(5):16-23, May 1982.
- [2] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software Practice & Experience*, 13:671-685, 1983.
- [3] Paul N. Hilfinger and Phillip Collela. FIDIL: A language for scientific programming. Technical Report UCRL-PREPRINT 98057, Lawrence Livermore National Laboratory, January 1988.
- [4] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235-246, Austin, Texas, August 1984.
- [5] Fred Obermeier and Randy Katz. EPOXY: An electrical and physical layout optimizer that considers changes. Technical Report UCB/CSD 87/388, UCBCS, November 1987.
- [6] John Ousterhout. A switch-level timing verifier for digital MOS VLSI. *IEEE Transactions on CAD*, CAD-4(3), July 1985.
- [7] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

and decreased the memory consumption of the program by 13%.

A RISC Approach to Runtime Exceptions

Mark Himmelstein
MIPS Computer Systems, Inc.
930 Arques Ave, Sunnyvale, CA 94086
himel at mips.com

Steven Correll
Key Computer Laboratories
6681 Owens Dr, Pleasanton, CA 94566
{sum,pyramid}!pacbell!master!sjc

Kevin Enderby
NeXT, Inc.
3475 Deer Creek Rd, Palo Alto, CA 94304
enderby at NeXT.com

Abstract

Moving cost and complexity from execution-time to compilation-time is a common theme among compilers for RISC architectures. We describe a facility for handling runtime exceptions which, by following that theme, costs nothing in execution speed until a program actually raises an exception. The facility was also easy to implement, because it builds upon mechanisms which already existed in our compiler system and in Unix.¹

1. Introduction

A runtime exception like division by zero is an unusual event. Many languages permit the programmer to provide a procedure, called an *exception handler*, which will execute when the exception happens. The handler may print an error message or alter a global data structure, and it may also execute a non-local "goto" to return to an outer-level procedure to recover quickly from the error.

Typically, each ordinary procedure of a program may provide a different handler, and if a particular procedure fails to provide one, it inherits a handler from the procedure which called it. That procedure may in turn have inherited the handler from *its* caller, and so on.

Inheriting a handler is a simple concept which poses an interesting implementation problem, because a particular procedure may be called from many different places in a program, and may therefore use any of a number of different handlers. Further, an exception may occur at a point in the program where a highly optimizing compiler has, unbeknownst to the handler, allocated variables to registers.

Therefore, when an exception occurs and the current procedure has not specified a handler, the runtime system must be prepared to *unwind* the procedure-call stack: it must methodically undo the effect of each procedure call, restoring registers and popping stack frames, until it reaches the desired outer-level procedure which provides the handler. Thus it needs to know the size of each frame and which registers were saved (because an optimizing compiler saves as few registers as possible in each frame).

The procedure-calling mechanism for a CISC (complex instruction set computer) invariably provides a frame pointer, which together with the stack pointer indicates the frame size. Often, the stack frame contains a bit-mask indicating which registers were saved. Sometimes, another bit-mask or pointer indicates exception handlers. This makes unwinding and exception-handling easy, but imposes on every procedure call the expense of updating the frame pointer and manipulating the bit-masks, even if the program never encounters an exception. [Dec81]

When we set out to add exception handling to the compiler system for the MIPS R2000² RISC (reduced instruction set computer) processor [Chow86, Mouss86], we had already defined a common stackframe convention for all languages. To reduce the cost of a procedure call, this convention eliminates the use of frame pointers and register-masks wherever possible: the frame size and the list of saved registers are placed in the symbol table at compilation time, not in registers at execution time; when, for example,

¹Unix is a registered trademark of AT&T.

²R2000 is a registered trademark of MIPS Computer Systems, Inc.

the debugger needs to trace the stack, it obtains a "virtual frame pointer" from the symbol table rather than an actual frame pointer from a register. We manifest the frame pointer in a physical register only within a procedure which actually performs dynamic allocation by altering the frame size during execution.

To keep procedure calls inexpensive, we took as our starting point the mechanism already used by the debugger to cope with the absence of a frame pointer.

2. Our Goals

Our initial goals were:

- (1) Because exceptions are not the rule, exception-handling should cost nothing in execution speed until you actually use it.
- (2) Raising an exception should be as fast as possible without violating goal 1.
- (3) Language-specific exception handling should mesh smoothly with the *signal* exception mechanism provided by the Unix operating system (to facilitate this, we added to our System V implementation certain features from our BSD implementation).
- (4) The exception mechanism should be versatile enough to handle the varying semantics of Ada [ADA83], PL/I [PLI81], and IEEE floating point [IEEE85], so that we could share implementation costs, and so that the various flavors of exceptions are all available to the user who links code from multiple languages into a single program.

3. Our Strategy

The rules for exceptions vary considerably among languages. An Ada exception handler may not resume execution at the point where the exception was raised; a PL/I or IEEE handler may. PL/I handlers obey dynamic scope rules; Ada and IEEE handlers obey a combination of static and dynamic rules.

To cope with this, we provide both a universal exception dispatcher and language-specific dispatchers. The universal dispatcher unwinds the stack until it finds an appropriate place to pass control to a language-specific dispatcher, which finds and invokes the appropriate user-written handler.

3.1. Finding the Right Handler

The universal dispatcher is driven by an *exception_info* data structure, initialized by the language translator at compile time and placed in the data space of the executable program by the linker. If a user procedure named "p" provides exception handlers, the translator constructs an array of these records to describe the handlers and names it "p_exception_info":

```
typedef struct exception_info {
    long    exception; /* what kind of exception */
    void    (*handler)(); /* handler for this kind of exception */
    long    data; /* compile time data to pass to handler */
} p_exception_info[ ];
```

The *exception* field indicates which kind of exception each handler deals with:

```
#define EXC_END 0 /* end of array for this proc */
#define EXC_BASE 1000000 /* start of EXC constants */
#define EXC_ALL (EXC_BASE+0) /* handles any exception */
#define EXC_ADA_USER (EXC_BASE+1) /* ada user exception */
#define EXC_PL1_USER (EXC_BASE+2) /* pl1 user exception */
```

The *handler* field could, in simple cases, point directly to a user-coded handler, but in most cases will point to a language-specific dispatcher. The *data* field lets the translator provide an additional word of data to be passed to that dispatcher at runtime if needed.

The interface to the universal dispatcher itself is:

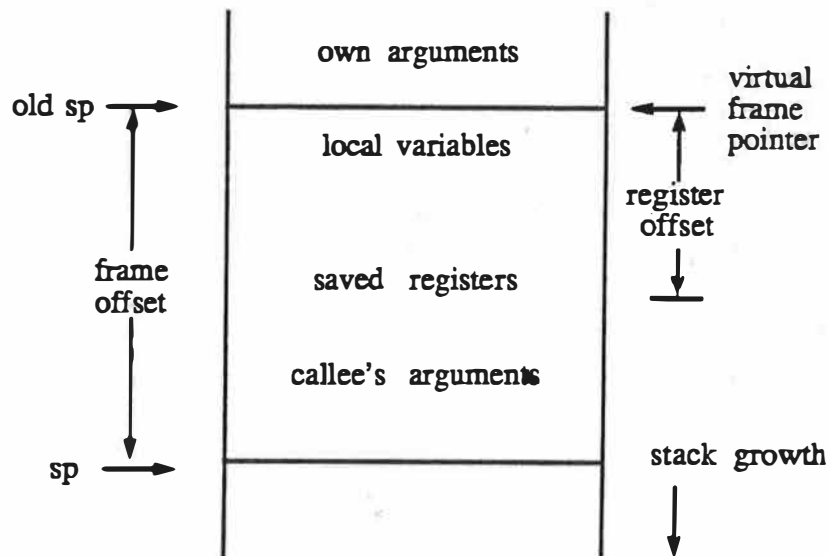
3.2. Unwinding the Stack

Within the universal exception dispatcher, the process of unwinding the stack requires a second data structure called the runtime procedure descriptor table or *rpd*. This is initialized by the linker on demand, sorted according to the starting address of each procedure, and stored in the data space of the executable program:

```
typedef struct runtime_pdr {
    unsigned long adr;          /* memory address of start of procedure */
    long regmask;              /* save register mask */
    long regoffset;            /* save register offset */
    long fregmask;             /* save floating point register mask */
    long fregoffset;           /* save floating point register offset */
    long frameoffset;          /* frame size */
    short framereg;            /* frame pointer register */
    short pcreg;               /* register or frame offset of return pc */
    long irpss;                /* index into the runtime string table */
    long reserved;
    struct exception_info *exception_info; /* pointer to exception array */
};
```

The *irpss* field is an index into a linker-created string table containing procedure names, useful for printing stack traces or runtime diagnostic messages. The linker makes the *exception_info* field for each procedure "p" point to the "p_exception_info" array described previously.

To understand the other fields, consider the MIPS stack frame format:



A procedure's prologue pre-allocates enough stack space for local variables, saved registers, and the largest argument-build area required for any called procedure, and the code usually accesses these via the stack pointer *sp*. To find the value of the virtual frame pointer *vfp* (which can be used like a conventional frame pointer), one must add the *frameoffset* field of the *runtime_pdr* to the contents of the register specified by the *framereg* field, which is usually *sp*, but which designates a true frame pointer register if a procedure requires dynamic local allocation. To find the saved registers, one adds the *vfp* and the *regoffset*. *Regmask* dictates which registers need saving or restoring and *pcreg* designates the register containing the return address (in leaf procedures, the compiler refrains from saving this in the stack).

The basic algorithm for unwinding is:

```
void
exception_dispatcher(long exception, long code, struct sigcontext *context);
```

Aficionados of BSD Unix will recognize this as the handler interface defined by the *sigvec* system call: *sigcontext* is the standard BSD Unix data structure describing the process state (e.g. the values in the register set, the program counter, etc.) and *code* is a constant which may designate a sub-category of *exception*. Our algorithm does not mandate this interface, but when an exception results from a Unix signal like SIGFPE (floating point exception), it is convenient to be able to pass *exception_dispatcher* directly to the *sigvec* system call. To facilitate this, we made our EXC constants distinct from Unix signal numbers.

A typical exception starts with a call to *exception_dispatcher*, either directly or as a result of a signal from the operating system. It then unwinds the stack, searching through the *exception_info* table to find and invoke the right language-specific dispatcher (invoked as *p->handler* in the following pseudocode):

```
exception_dispatcher(exception, code, context)
    long exception;
    long code;
    struct sigcontext *context;
beginproc
    struct sigcontext raised_at;
    struct exception_info *a;

    raised_at = context
    current_address = context.pc
    loop while current_address != 0 do
        search for an exception_info array "a" for the procedure
            containing current_address

        if found then
            loop through a
                if a->exception == EXC_END then
                    break
                elseif a->exception == exception or a->exception == EXC_ALL then
                    call p->handler(exception, code, raised_at, context);
                    break
                endif
            endloop
        endif

        unwind stack, changing context to that of preceding frame
        current_address = context->pc;
    endloop

    call "last chance" exception handler provided by the language of
        the main program
endproc
```

Notice that the language-specific dispatcher *p->handler* receives both the *sigcontext* of the procedure which raised the exception and that of the procedure which provided the handler for it, plus the *exception* and *code* arguments.

After executing, *p->handler* has two choices:

1. It may return to the exception dispatcher, allowing it to continue unwinding the stack, possibly invoking additional handlers in outer procedures.
2. It may stop the unwinding by calling Unix *longjmp* or *sigreturn*. This may perform a nonlocal goto to an outer procedure or (using the *raised_at* sigcontext) may resume executing the code which originally raised the exception.


```

unwind(context)
  struct sigcontext *context;
beginproc
  find runtime_rpd corresponding to context->pc
  vfp = context->reg[runtime_rpd.framereg] + runtime_rpd.frameoffset;

  /* restore registers */
  loop through runtime_rpd.regmask
    if bit set in runtime_rpd.regmask then
      regaddr = vfp + runtime_rpd.regoffset + (bit index in regmask)
      context->reg[(bit index in regmask)] = *regaddr;
    endif
  endloop

  do the same for floating-point registers

  /* get new pc from return register */
  pc = context->reg[runtime_rpd.pcreg];
  /* restore sp to vfp */
  context->reg[sp] = vfp;
endproc

```

Note that this is a “virtual” unwind, which operates on the *sigcontext* data structure rather than the actual machine. One may pass that data structure to a Unix *longjmp* or *sigreturn* system call to change the real machine state to match that specified by the data structure.

3.3. Runtime Costs

Only two aspects of this algorithm are costly: searching the register masks to restore registers, and mapping the current pc to the *rpd* for the procedure containing it.

Register masks are sparse. For example, the p11-coded p11 translator contains 528 procedures with a (static) average of 1.7 registers saved per procedure. Rather than shift, mask, and compare 63 times for the 63 bits (representing both general purpose and floating-point registers) within the masks, we successively split the mask in half down to the byte level till we find a nonzero half, and then extract bits. Though the worst case requires 14 comparisons and 32 bit extractions, the best case requires but one comparison, and the average is closer to the best case.

Mapping from a pc to the procedure containing it is complicated by the fact that the *rpd.adr* table entries bound the pc rather than matching it; unless the pc happens to match a starting address, a binary search will cost $\log(n + 1)$ probes (the worst case) plus an extra check to see whether the pc lies above or below the last address probed. To speed this, we maintain a 128-entry exception cache. On a test of a program which contained 871 procedures and which raised the same exception 100 times, adding the cache reduced the cost from 11 probes ($\log(871+1) + 1$ (for the fall-out check)) to 1.12 probes.

4. Three implementations

To make the preceding description clearer, we'll discuss three specific implementations based on this general mechanism.

4.1. Ada

The Ada language completely defines at compilation time how to handle an exception for every instruction in the program. An Ada exception handler may only appear at the end of its scope. It is non-resumptive, and so cannot return to the code which raised the exception; but it may fall out of the scope, return, or *raise* the exception to an enclosing scope.

Except for two problems, we could eliminate the Ada language-specific handler and set the *handler* field of each *exception_info* record to designate the appropriate handler. The first problem is that within a procedure, the programmer may define a new scope with its own handler of the form:

```

begin    -- start of scope
--
-- user statements
--
exception
  when exception_name_x, exception_name_y, ... =>
    -- handler statements
    --
  when exception_name_a, exception_name_b, ... =>
    -- handler statements
    --
...

end;     -- end scope

```

The second problem is that an exception in one scope may be reraised first to any statically enclosing scope, and then to any dynamically enclosing scope; in fact, this is the default whenever an exception is raised in a scope which did not specify a handler.

Our strategy is to let the universal exception dispatcher find the proper dynamic scope, and let an Ada-specific dispatcher either find the appropriate static scope at that level, or return to the universal dispatcher for further unwinding.

In our compiler, which uses the VADS³ front-end, each statement of the program belongs to a *handler group*, a group of statements which share a common set of exception handlers; and the compiler emits statements in an order which strictly reflects the static nesting of scopes. For example:

Handler Group	Statement Number	Ada statements
1	1	begin
2	2	begin
1	3	-- dummy inserted by compiler
3	4	
2	5	exception
2	6	when x,y =>
2	7	
2	8	end;
2	9	begin
1	10	-- dummy inserted by compiler
4	11	
2	12	exception
2	13	when a =>
2	14	
2	15	when b =>
2	16	
2	17	end;
1	18	end;

The handler groups for that code segment are:

Group	Name	Handler start statement number
3	x	7
	y	7
4	a	14
	b	16

³VADS is a registered trademark of Verdex Corporation.

Groups lacking explicit handlers are omitted and turned into *reraises*. Each time a handler-group number changes, a frame structure is created. Each frame contains the beginning and ending addresses (we'll use statement numbers here for clarity) and a pointer to the appropriate handler group. So in this case, our frame structure table is:

Beginning Statement	Ending Statement	Handler Group
1	1	1
2	2	2
3	3	1
4	4	3
5	9	2
10	10	1
11	11	4
12	17	2
18	18	1

If a particular frame does not handle a given exception, the runtime system should merely check the next frame; when it passes the outermost lexical scope (i.e., frame 1), it should then check the next dynamically enclosing scope. The table is actually compacted by removing any entry which can only result in unwinding the stack to check the next dynamic scope:

Beginning Statement	Ending Statement	Handler Group
4	4	3
11	11	4

The absence of a frame indicates the need to unwind, and so does a discontinuity in statement numbers (actually, instruction addresses).

The translator divides the work between the universal dispatcher and the Ada-specific one by constructing a one-element "p_exception_info" array for each procedure "p" with the language-specific *data* field pointing to the appropriate frame table for that procedure:

```
struct exception_info {
    long exception;
    void (*handler)();
    long data;
} p_exception_info = { EXC_ALL, Ada_handler, (long) &p_frame_table };
```

Every Ada procedure uses the same *Ada_handler* routine, which searches the specified frame table according to the semantics just described. It can either invoke the appropriate user-written handler, or return to the universal dispatcher to unwind to the next dynamic scope.

4.2. PL/I

A PL/I program installs an exception handler by executing an ON statement for the appropriate class of exception, such as "zerodivide":

```
ON zerodivide BEGIN;
    got_zerodivide = 'l'b;
    goto retry;
END;
```

(Thus PL/I dictates a slight revision of our earlier goal: the exception mechanism is free provided you never encounter an exception or execute an ON statement.)

Handlers are dynamically scoped: a handler remains in effect until the procedure which installed it returns, but a child procedure can temporarily override a parent's handler by executing its own ON statement for the same class, which remains in effect until the child returns.

After executing, a handler may perform a nonlocal goto, or return to the point at which the exception occurred.

To implement this, the *PL/I ON* statement dynamically allocates a descriptor within the stack frame of the containing procedure (thus guaranteeing that when the procedure returns, its handlers get uninstalled for free). A pointer within the stackframe indicates the first descriptor, and descriptors for additional classes are linked onto it as a list.

```
/* (Simplified) descriptor for each instance of the PL/I ON statement */
struct ob_struct {
    struct ob_struct *prev;          /* NeXT block in the chain */
    unsigned int handler_pc;         /* Address of handler */
    unsigned int condition;          /* Class of exception */
    unsigned int current_fp;         /* Frame ptr for handler */
};
```

For each procedure containing an *ON* statement, the *handler* field of the *exception_info* structure described earlier points not to an actual handler, but rather to a runtime routine called "*V\$PL1_DISPATCHER*."

Procedures not containing *ON* statements are treated as if no exception mechanism existed, and incur no cost.

When an exception occurs, the universal dispatcher unwinds the stack until the *exception_info* structure tells it to call *V\$PL1_DISPATCHER*, which traverses the list of descriptors looking for a suitable handler.

```
V$PL1_DISPATCHER(exception, code, scp, oscp, this_proc)
    unsigned exception;              /* Always EXC_PL1_USER */
    unsigned code;                   /* Which class of exception */
    struct sigcontext *scp;           /* Frame which raised exception */
    struct sigcontext *oscp;          /* Frame containing ON stmt */
    struct runtime_pdr *this_proc;    /* Pdr info about oscp */

beginproc
    temp = use oscp and this_proc to find pointer to head of list;
    loop while temp != NIL
        if temp->condition == code then
            invoke temp->handler_pc with $fp = temp->current_fp;
            use Unix "sigreturn" system call to return to frame scp;
        endif
        temp = temp->prev;
    endloop
endproc
```

If successful, *V\$PL1_DISPATCHER* invokes the handler, making it appear to have been called from the code which raised the exception (we take advantage of the "virtual" nature of the unwinding, so that the actual stack is unchanged and the handler may simply return if desired). If unsuccessful, *V\$PL1_DISPATCHER* returns to the universal dispatcher, which continues unwinding till the next frame which contains *ON*-statement descriptors.

4.3. IEEE floating point

The IEEE floating point standard specifies five different kinds of exceptions: invalid, divide by zero, overflow, underflow, and inexact. The user may provide a handler for each of these; may disable, save, or restore an existing handler; and may ask whether a handler is enabled.

The handler must be able to determine the nature of the exception, the arithmetic operation, the destination format (e.g., single or double precision), the correctly rounded result (for overflow, underflow and inexact), and the operand values (for invalid or divide by zero). And the handler must be able to return a value for use in lieu of the normal result.

The standard says that it is acceptable to implement some operations (trigonometric functions, for example) as subroutine calls rather than individual hardware instructions, and in fact the MIPS R2000 hardware does so. However, this must be invisible to the user. Such a subroutine must still behave like a single instruction in the sense that when an instruction within that subroutine raises an exception, the user's handler can repair the result of the subroutine, not just the result of that instruction. Thus, for

example, the handler must be made to believe that "tan," rather than a multiplication within the "tan" subroutine, raised the exception.

The IEEE standard does not define any language syntax with which to implement these suggestions. At MIPS, we propose that the user be allowed to specify at compile-time a set of handlers for each procedure. If a procedure has no handler for a particular exception, it can inherit one from a dynamically enclosing procedure. A procedure may call a builtin routine to enable or disable any handler, or to obtain a pointer to each of the enabled handler procedures.

Ultimately, we would like to provide pragmas in C, Pascal, and Fortran, so that (in the following example) procedure "p" would invoke user-written procedures called "invalid," "divide0," "overflow," "underflow," and "inexact" to handle the corresponding IEEE exceptions:

```
#pragma fp_handlers(p, invalid, divide0, overflow, underflow, inexact)
```

Naturally, one could specify an incomplete set of handlers as:

```
#pragma fp_handlers(p, invalid, , overflow, , inexact)
```

Meanwhile, we have prototyped a preliminary implementation using C preprocessor macros to generate the code which the language translators would eventually emit in response to the pragmas. The macros are:

FP_INIT, which must be placed in the main program, initializes the exception-handling mechanism.

FP_HANDLER associates user-written handler procedures with a specified procedure in the program.

FP_ENABLE and **FP_DISABLE** enable and disable hardware exceptions. (Our floating-point hardware runs faster with exceptions disabled because it can execute instructions in parallel without fear of imprecise exception behavior. Thus, in the pragma-based implementation, the compiler would emit code to enable exceptions at the beginning of each procedure which specifies handlers, and to restore the previous state of the enable-flags on return.)

The user-written handler receives from the IEEE runtime system the following data structure, and can alter the result of the interrupted operation by merely storing into the *dest* field before returning:

```
struct handler_info {
    long fpc_csr;      /* floating point control and status register */
    long instr;        /* machine instruction (if non-zero) */
    char *routine;     /* routine name (if instr is zero) */
    long src1_fmt;     /* source 1 operand's formats */
    long src2_fmt;     /* source 2 operand's formats */
    long dest_fmt;     /* destination's format */
    union value src1;  /* source operand 1's value */
    union value src2;  /* source operand 2's value */
    union value dest;  /* destination's default value */
};

union value {
    struct {
        unsigned long v0,v1,v2,v3;
    } v;                /* four words for the value */
    long w;             /* word */
    float f;            /* single floating point */
    double d;           /* double floating point */
    long b;             /* boolean (0 == FALSE, otherwise TRUE) */
};
```

To understand how our IEEE exception facility uses the universal exception mechanism described earlier, consider the pragma or the equivalent macro:

```
#pragma fp_handler(p, , myhandler, , , )
FP_HANDLER(p, 0, myhandler, 0, 0, 0)
```

either of which specifies *myhandler* as the handler for divide-by-zero exceptions in the routine “p.” The translator builds a two-element *exception_info* array:

```
typedef struct exception_info {
    long exception;
    void (*handler)();
    long data;
} p_exception_info[2] = {
    { SIGFPE, ieee_sigfpe, (long) &p_fp_handlers },
    { EXC_END, 0, 0 }
};
```

which tells the universal exception dispatcher to hand control to the IEEE-specific dispatcher “ieee_sigfpe” when a Unix SIGFPE (floating point exception) signal occurs. The IEEE-specific dispatcher must choose the appropriate user-written handler, set up the *handler_info* structure, and pass it to the handler. When the handler returns, the dispatcher is also responsible for replacing the destination value and resuming execution.

To drive the IEEE-specific dispatcher, the translator creates the secondary data structure *p_fp_handlers*:

```
struct fp_handlers {
    long enables;
    long old_enables;
    int (*handlers[5])();
} p_fp_handlers = {
    0,
    0,
    (int (*)())0,
    (int (*)())myhandler,
    (int (*)())0,
    (int (*)())0,
    (int (*)())0
};
```

The *enables* and *old_enables* fields provide room to save and restore the hardware enable-flags on entry to and exit from the routine “p”; the *handlers* array tells the dispatcher which user-written handlers to invoke. In the absence of a handler for the appropriate exception, the IEEE-specific dispatcher returns to the universal dispatcher, which unwinds the stack until it reaches another procedure containing an *exception_info* array, and then invokes the IEEE-specific dispatcher once again.

5. Multiple Languages in a Single Program

Though we have described the three implementations separately, they can within limits coexist in a single program, because the universal exception dispatcher is quite neutral: it merely unwinds the stack until it finds an *exception_info* entry whose *exception* field is appropriate, and then invokes the dispatcher indicated by the *handler* field. Suppose that a PL/I runtime raises the CONVERSION exception. The universal dispatcher will unwind the stack until it finds an *exception_info* entry whose *exception* field is EXC_PL1_USER, and will then invoke “V\$PL1_DISPATCHER.” In the process it may have to unwind past Ada-coded procedures which specify their own exception handling mechanism. These pose no difficulty, because their *exception* fields contain EXC_ADA_USER instead.

6. Conclusion

Using compile-time complexity to improve execution-time speed has been a guiding principle of the MIPS compiler system from start. In this instance, it led us to an exception mechanism which not only provides high performance for current languages, but also promises to expand to fit the needs of future languages. And while a RISC architecture certainly encourages such a lean and spare approach, other architectures could in fact profit from it, too.

7. References

[ADA83]

"Ada Programming Language," ANSI/MIL-STD-1815A-1983.

[Chow86]

Fred Chow, Mark Himmelstein, Earl Killian, and Larry Weber, "Engineering a RISC compiler System," *Proceedings COMPCON*, IEEE, March 4-6, 1986.

[Dec81]

Digital Equipment Corporation, "Vax Architecture Handbook," 1981

[Mouss86]

John Moussouris, Les Crudele, Dan Freitas, Craig Hansen, Ed Hudson, Steve Przybylski, Tom Riordan, and Chris Rowen, "A CMOS RISC Processor with Integrated System Functions," *Proceedings COMPCON*, IEEE, March 4-6, 1986.

[IEEE85]

"IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985.

[PLI81]

"American National Standard Programming Language PL/I General-Purpose Subset," ANSI X3.74-1981.

CASPER the Friendly Daemon

*Ronald E. Barkley
AT&T Information Systems
Summit, New Jersey
attunix!rebark*

*Danny Chen
AT&T Bell Labs
Holmdel, New Jersey
hocus!dwc*

Abstract

Despite the fact that timing and tracing studies provide tremendous information on the dynamic behavior of software systems, there has been surprisingly little support for these activities under UNIX® System V. In this paper, we present a set of tools, called CASPER, packaged as a C library and a pseudo-device driver for UNIX System V, which support timing and tracing of software at both the kernel and user level. Key features of this package are its extreme ease of use, its efficiency, and its ability to time and trace both user and kernel level activities into the same log file. To illustrate its ease of use, we provide some "real life" examples of tracing studies using CASPER.

1. Introduction

Performance measurement and analysis is playing an increasingly important role in the development cycle of software systems and applications. Under UNIX System V, there are a number of tools and methods available to measure, evaluate, and analyze the performance of both the system and applications.

For example, one can look at UNIX process accounting to provide information on the CPU and memory requirements of a program.^[1] One can use profiling to obtain a function level breakdown of CPU usage. The same technique can be used to measure function level CPU requirements at the kernel level.^[1] Weinberger's instruction counting method^[2] can be used to examine the frequency of execution of *basic program blocks*.¹ The System Activity Report (SAR) Package provides system-wide statistics on the consumption of various system resources.^[3] And if one has access to a standalone system, one can use SAR to measure the system resource requirements of a particular workload.

These tools can be classified as providing aggregate, summary measurements. To gain further insight into the behavior of systems and applications, however, it is often necessary to obtain information on their time-varying characteristics. Such information often requires a trace of the execution of the system under examination. In this paper, we describe a package that supports timing and tracing activities under UNIX System V. This package, which we call CASPER, provides a high level interface to most of the services required to trace software at both the user and kernel levels.

1. A *basic program block* is a block of code in which, once entered, all instructions in the block are executed.

2. CASPER History

CASPER began as a specialized set of kernel modifications designed to collect page fault traces on demand paging versions of UNIX System V.² The "guts" of this page fault trace tool were then taken and adapted to collect traces of other system events. Having a basic timing and tracing system to modify and adapt to the needs of specific studies greatly facilitated the progress of these studies. After several customizations of this tracing package, the authors decided to redesign these ad hoc modifications into the generic timing and tracing package we are describing in this paper.

3. CASPER Architecture and Implementation

From our experiences with the original ad hoc kernel instrumentation package, we identified certain essential services that the system should provide to support tracing. These services are

- Timestamping
- Data buffering
- Data storage
- Data retrieval

Developing and debugging these operations can take a large portion of the time needed to obtain a trace if these operations have to be developed from scratch. By providing a high level interface to these services, CASPER allows the instrumenter to obtain data in an hour's time instead of days.

In this section we will describe the architecture of CASPER and how it provides these tracing services. To better explain our design decisions, we describe the architecture in the context of performing kernel level instrumentation.

3.1 Timestamping and Tracing at the Kernel Level

3.1.1 Generating High Resolution Timestamps. One of the more common pieces of state information needed in tracing events is the time of the occurrence of the event. Under UNIX System V, there are a number of different concepts of time. There is the concept of *virtual time* for processes that is a count of the number of clock ticks of CPU time that has been charged to a process.³

There is also the concept of *wall clock time* that is represented by the global system variable `time`. `Time` counts the number of seconds that have elapsed since system boot. A higher resolution wall clock time is available through the global system variable `lbolt`.

Although the resolution of the `lbolt` timer is fairly high (typically either 1/60th or 1/100th of a second), we felt that this was not a high enough resolution for many measurement purposes. Fortunately, many systems have access to a higher resolution clock that is not interrupt driven but must be polled for its value. On the AT&T 3B2, this clock is the programmable interval timer that is used to generate clock interrupts. This clock, which is loaded with an initial value that it decrements at a 0.1MHz rate, and which generates a clock interrupt when it reaches zero, can be accessed through a memory mapped address. Since the value of this timer is the number of tens of microseconds until the next clock interrupt, and since `lbolt` is a count of the number of clock interrupts since system boot, CASPER combines these two values into a counter of the number of tens of microseconds since the system boot.

Because of the machine dependent nature of this timestamp (in both its generation and its resolution), CASPER provides this timestamping service automatically when event records are saved. The generation of this high resolution timestamp is the only machine dependent portion of CASPER. We

2. Developed by Ron Barkley, Paul Lee, and Hira Nirmal.

3. This measure is sampled by the clock interrupt handler. Whichever process is executing when the clock interrupt occurs is charged for the entire clock tick. This measure is further discriminated by whether the process was in user or system state when the clock interrupt occurs.

leave it up to the instrumenter to include other measures of time (e.g. virtual time) as part of their user defined records since these values are easily obtainable.

3.1.2 Data Buffering. Having decided what events to log, the instrumenter must allow for the buffering of these event records. There are three reasons why buffering of these event records is both desirable and necessary:

1. We would not want to incur a single I/O for each record that must go out to secondary storage.
2. Events might occur at a faster rate than can be accommodated by the I/O subsystem.
3. (Most importantly) The UNIX operating system does not allow one to do I/O at arbitrary points in the kernel.

Thus CASPER manages a circular buffer in kernel memory. This buffer, whose size is user configurable, is broken up into several, smaller, logical buffers, whose size is also configurable.⁴ These logical buffers allow us to implement a simple flow control policy. When the buffering of an event record crosses a logical buffer boundary, an I/O is initiated on any full logical buffers that have not yet been written out. By varying the sizes of the circular buffer and the logical buffers (and thus the number of logical buffers), we can accommodate events whose occurrences are bursty in nature.

Access to this buffer is provided through any one of three CASPER kernel level routines: *casrecord()*, *casword()*, and *casfword()*. *Casrecord()* takes as arguments a pointer to a user defined record and the size of that record. It generates the high resolution timestamp, places this timestamp onto the end of the circular buffer, copies the record onto the end of the circular buffer and arranges for I/O if a logical buffer boundary is crossed. *Casword()* is very similar to *casrecord()* except it takes a single argument that is a word to be timestamped and saved. *Casword()* is provided to avoid the overhead of the general memory copy routine used to copy variable size records. *Casfword()* is a macro that is expanded inline by the preprocessor and is provided to avoid the overhead of calling a function to do the timestamping and circular buffer management. It takes two arguments: the word to be timestamped and saved, and a *trace level*. This *trace level* is provided for greater control of the instrumentation process in the following manner.

Kernel level instrumenters are asked to follow a certain convention in using CASPER. There is a global kernel variable, *traceflg*, used by CASPER that represents the trace level. At the minimum, instrumenters are asked to test if (*traceflg* > 0) before calling any of the above data buffering functions. This minimum requirement allows us to control the starting and stopping of CASPER services. Finer control can be obtained if the instrumenter wished to use different *traceflg* values to control different measurement functions. For example, s/he might instrument the system to collect two different types of events. The trace level can then be used to control which of these two events to log (or maybe a third value to indicate that both types of events should be logged).

3.1.3 Data Storage. As mentioned in the previous section, we would like a logical buffer to be written out to secondary storage whenever a logical boundary is crossed. However, we also know that since the event record that resulted in a crossing of a logical buffer can occur at almost any place in the kernel, there may be situations in which we cannot do I/O when that boundary is crossed. In addition, the delay to a process in doing an I/O might greatly perturb the real time measurements of a process.

As a result, CASPER uses a system level process, named *casper*, to do the I/O. *Casper*, which is spawned dynamically only when timing and tracing are turned on, sleeps on the address of the start of the circular buffer. When a logical boundary is crossed, a wakeup on that address is performed. *Casper* then looks at the last logical buffer that was written out and loops until all full logical buffers have been written. It then goes to sleep again until the next wakeup.

4. The only restriction is that the size of the entire circular buffer be an exact multiple of the size of the logical buffer.

Other approaches to saving instrumentation data to files have included the implementation of a system call to allow a user level process to read the data from the circular buffer. A user level daemon process then continually loops and uses this new system call to read this data from the circular buffer and write the data out to a file.⁵ This approach is functionally equivalent to our approach of using a kernel level daemon except that it requires a copy of the data from kernel space into user space for the process to read it and then requires another copy of the same data (i.e. no modifications) back into kernel space to write the data out to file. Using a kernel level daemon also allows us greater control on the amount of time to react to full buffers.

One final point on the kernel level implementation remains. There are times when the anticipated volume of data is great and we would like to direct the event log to a raw device such as a streaming tape or a floppy or even a raw hard disk instead of the filesystem. However, in UNIX System V, there is a restriction that requires that the data for *physios* originate from user space. Rather than write our own version of *physio*⁶ for CASPER, we found the following solution.

We use the region architecture that is made available to us under the demand paging versions of UNIX System V⁽¹⁾ and allocate and attach a region that lies in user space for the casper daemon. We then *map* the pages of this region to be the page frames of the circular buffer that resides in kernel space. Thus, although the actual data that resides in the page frames of the circular buffer is accessible to all kernel level routines (via *casrecord()*, *casword()*, and *casfword()*), casper can read this data through the region that is mapped in user space. *Physio()* does not complain since the data is coming from user space.

3.2 User Level Architecture

All of the timestamping, data buffering and data logging features of CASPER have been implemented at the kernel level. The user level has been given control of the execution of CASPER. The user level also has access to the timing and tracing services of CASPER. In this section we describe the user level architecture and how CASPER provides a general purpose interface to the services described earlier.

3.2.1 CASPER As a Special Character Device. CASPER is implemented as a device driver. This allows us to distribute the package separately from the UNIX kernel. Access to the various CASPER mechanisms is available to the user level through operations on the file */dev/casper* that is configured as a special character file. This means that in addition to the kernel level functions mentioned earlier, we have implemented kernel level functions to support opening, reading, writing, and ioctling the */dev/casper* file.

To access these functions from the user level, one must first open the file */dev/casper* and obtain a valid file descriptor. Since */dev/casper* is a UNIX file, we can use standard file permissions to administer access rights to the CASPER facilities. Also, multiple processes are allowed to open */dev/casper* to obtain access to CASPER facilities. This file descriptor can then be passed to *ioctl*, *read*, and *write* system calls to control and obtain CASPER services. Table 1 contains a list of the various *ioctl* options and their effect on CASPER.

Through this interface, we have control over: the destination file of the trace data, the size of both the circular buffer and the logical buffers, the priority at which casper runs, the spawning and death of casper, and the trace level. For example, one goal in configuring CASPER is to never run out of empty logical buffers to place event records into.⁷ With events that occur in large bursts, we would have to increase the number of logical buffers and perhaps the size of the logical buffers. And while casper, by default, runs at the highest available priority to make sure logical buffers get written out promptly, in

5. See ⁽⁴⁾ and ⁽⁵⁾ for examples of this approach.

6. In order to keep the size of CASPER to a minimum, we use as many available kernel level routines as possible.

7. When an attempt is made to timestamp and log an event record when all logical records are full, a message is printed at the console, and tracing is disabled.

TABLE 1. CASPER ioctl Options

Option	Argument	Function Performed
C_BUFSIZ	(total buffer size, logical buffer size)	change size of total and logical buffers
C_PRI	UNIX priority	set the priority for casper
C_START	file descriptor for trace file	start casper
C_STOP	none	set trace level to zero and arrange for death of casper
C_TRACE	trace level	change trace level
C_UNTRACE	none	clear trace level

certain situations we might know that the event occurrence rate is low. We can then degrade the priority of casper to any valid UNIX priority.

We acknowledge that the ioctl interface may not be the most natural one to use. To simplify the use of CASPER, we have provided a higher level interface to these control functions in the form of a C library. Table 2 lists these functions and their effects.

TABLE 2. CASPER High Level Control Functions

Name	Function
casstart(filename, trace_level)	start tracing using <i>filename</i> for the trace file and <i>trace_level</i> for the trace level
casstop()	stop tracing
casbuffer(gbufsize, lbufsize)	set sizes of circular buffer and logical buffer
caspri(pri)	set priority for casper
casrecord(size, addr)	timestamp and save a record
casword(data)	timestamp and save a word
casimer()	return encoded high resolution timestamp
casdecode(ts)	decode a timestamp into 10s of microseconds

Finally, we include two user level commands, *casperon* and *casperoff*, which turn CASPER on and off.

3.2.2 Timing and Tracing at the User Level. As mentioned earlier, timing and tracing of user level software is a side benefit of developing CASPER as a device driver. In order to give user level processes access to the timing and tracing facilities of CASPER, we make use of the read and write operations on */dev/casper* in a natural way. A read of */dev/casper* simply returns the current value of the CASPER high resolution timer. A write to */dev/casper* arranges for the data that is passed to be timestamped and placed in the circular buffer.

Thus by simply relinking the operating system to include the CASPER device driver (i.e. without any modification of kernel source), we can time and trace user level software. More interesting applications of this feature are the tracing of entire systems where the interesting events are at both the user and kernel level (network protocols for example). This feature of CASPER allows us to obtain a single integrated trace of both levels.

3.2.3 Data Retrieval. One final service required by people doing instrumentation is the retrieval of the stored data. CASPER provides a library of user level functions to retrieve data from a trace file. Table 3 contains a list of these functions and what they do.

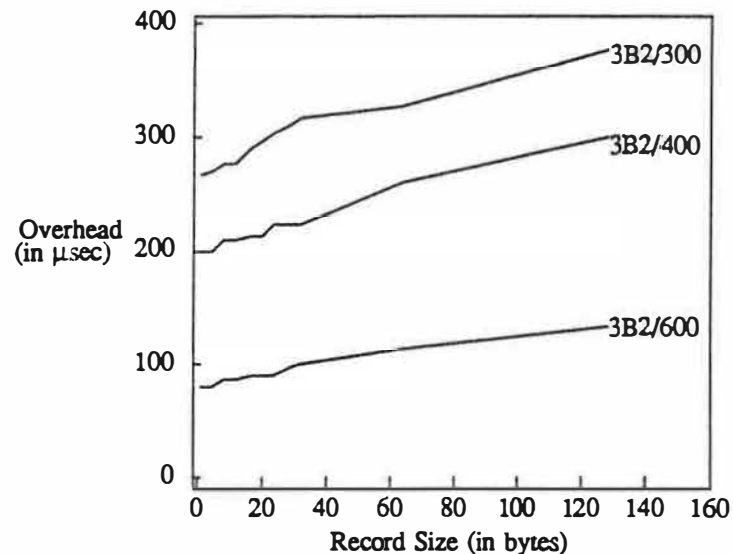
4. CASPER Overhead

We have attempted to reduce the overhead of using CASPER to a minimum. Because all of the data structures used by CASPER are dynamically allocated only when tracing is turned on, the memory overhead of CASPER, when linked into the system but not used, is just over 2Kbytes. When tracing is on, the memory overhead increases by the size of CASPER's circular buffer.

TABLE 3. Data Retrieval Functions

Name	Function
<code>casopen(filename, type)</code>	open <i>filename</i> as the tracefile
<code>casclose()</code>	close the tracefile
<code>casnext()</code>	go to next CASPER record
<code>casrtime()</code>	return the time field of next record
<code>casrsize()</code>	return the size of next record
<code>casrdata((char *) record)</code>	copy back next record into <i>record</i>

The following figures show the CPU overhead of using CASPER's different recording mechanisms. Figure 1 and Figure 2 show the overhead of using the CASPER routine `casrecord()` as a function of the size of data being saved. Figure 1 is applicable when using CASPER at kernel level, Figure 2 when at user level.

Figure 1. CPU Overhead for `casrecord` at Kernel Level

In both cases, the overhead is small enough for most applications. For the special cases that require minimal overhead and only want to save a single word of data, `casword()` and `casfword()`⁸ are provided. The overhead for using these is shown in Table 4.

Table 5 shows the cost in increased code size for every call to CASPER inserted in the code to be traced.

5. Examples

In this section, we describe some examples of the applications of CASPER. The first is a simple example of kernel instrumentation. The second is a more case involving more complex data to be collected. The third example will describe and illustrate a utility we have packaged with CASPER that allows us to do function level timing and tracing of C programs.

⁸ `Casfword()` is only available at kernel level.

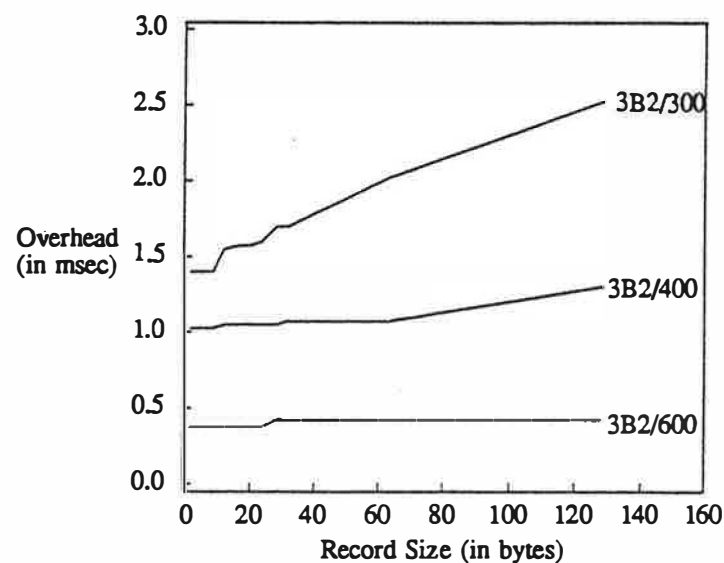
Figure 2. CPU Overhead for *casrecord* at User Level

TABLE 4. Overhead of Using Special CASPER Functions

3B2 Hardware	<i>casword()</i>		<i>casfword()</i>
	Kernel-level	User-level	
3B2/300	110 μ sec	1.21 msec	60 μ sec
3B2/400	80 μ sec	0.93 msec	40 μ sec
3B2/600	40 μ sec	0.50 msec	20 μ sec

TABLE 5. Overhead in Code Size for Using CASPER

CASPER Routine	Bytes per Invocation
<i>casrecord()</i>	20
<i>casword()</i>	20
<i>casfword()</i>	100

5.1 Process Switches

UNIX System V implements a round robin policy with a time quantum of one second for CPU scheduling. Most programs, however, tend to make many requests to the operating system which results in "voluntary" loss of the CPU. We were interested in finding the actual distribution of times that processes spend at the CPU.

In order to obtain this information, a single hook was placed in the kernel level routine responsible for arranging for the rescheduling of processes at the CPU. Since all we were interested in was the time of the process switch and the process id of the process that was getting switched out, we simply called *casword()* with the process id as an argument. The operating system was recompiled and a benchmark workload was run with CASPER turned on.

A user level program was written to examine the log file generated during the benchmark run and to print out the measured interprocess switch times. Figure 3 contains a histogram of the distribution of inter-process switch times that were measured for a certain multi-user benchmark. Using the high level kernel routines provided by CASPER, we were able to obtain this data (including histograms) within a hour of starting! (Of course, we knew exactly where we wanted to place our hooks.)

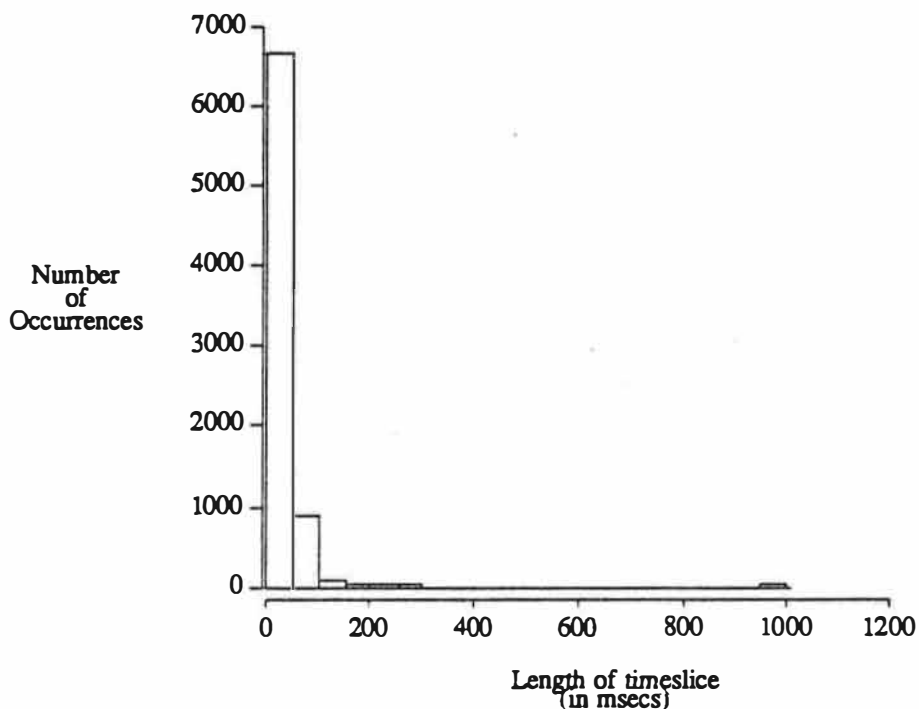


Figure 3. Distribution of Inter Process Switch Times

5.2 Memory Reference Sampling

Another interesting application of kernel instrumentation is in the sampling of memory references of processes. Taking an approach very similar to the one presented in [4], we simply had the clock interrupt handler invalidate the page table entries of the currently running process, arranged for another bit to keep track of whether a page was really valid, and inserted *casrecord* calls into the page fault handler to log some state information values at the time of the page fault.

By keeping track of the virtual address reference that generated the page fault, and the virtual process time at which the page fault occurred, we can produce memory reference charts of the type described in [4]. Figure 4 presents a sample of a memory reference chart for the text region of the 3B2 compiler.

Although the data acquired for this study was in some sense more complex than that of the process switch study, the data was just as easy to obtain using the high level routines provided by CASPER. Also, because of the way that we have designed CASPER, the high rate of events generated by the large number of pseudo-page faults posed no problem in terms of either CASPER's ability to keep up or overall system performance.

5.3 Function Level Timing and Tracing

Because the profiling of function calls is a common approach used in analyzing software systems, we provide some simple utilities with CASPER to help make such measurements. To time functions, we need simply insert calls to *casword* upon entering functions and before returning. The CASPER utility that we provide as part of the CASPER package, called *insert*, will parse a C source file, assign unique reference numbers to functions and maintain this mapping in a file that we call a *dictionary*, and insert *casword* calls at the beginning of, and before every return from, the functions in the file. Each of these calls passes a token to *casword* which identifies the function, and whether we are entering a function or if we are exiting it. Since there are multiple possible exit points from a function, each one is uniquely identified by the token.

As the traced functions are executed, CASPER collects and timestamps the tokens, placing them in the trace file. After the trace has been collected, we postprocess the trace file, identifying tokens, printing

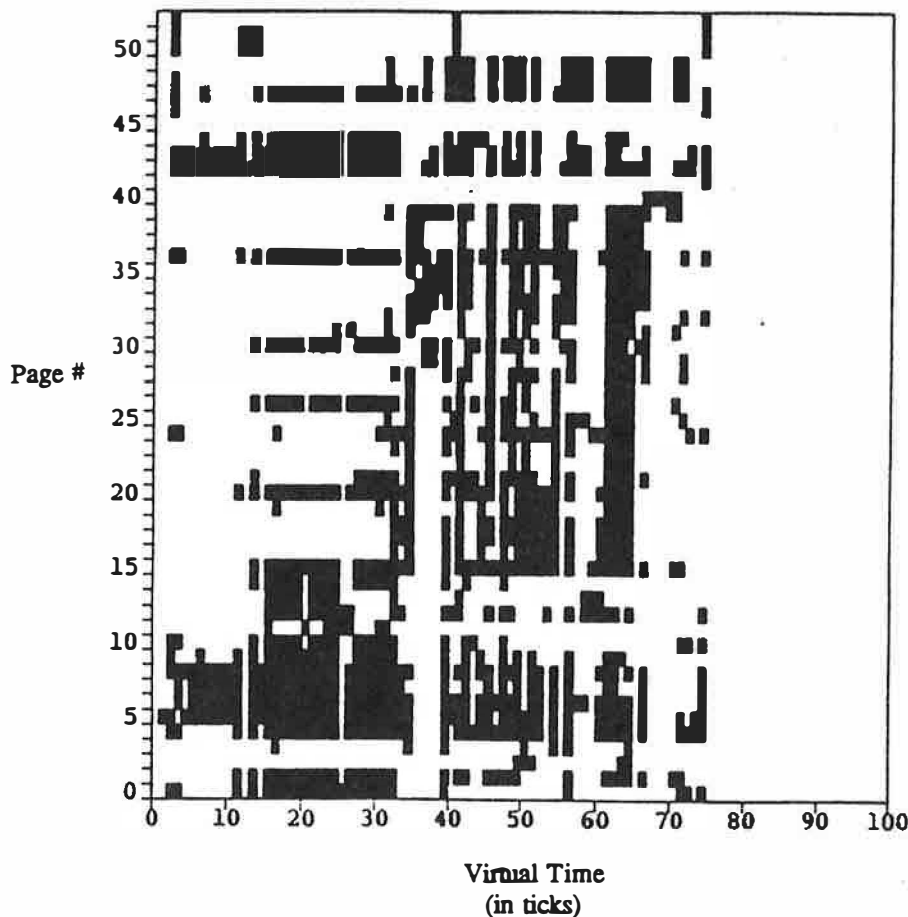


Figure 4. Memory Reference Chart

the sequence of function calls and returns and deriving the amount of time spent in each routine. We use the CASPER utility *prtrace* to unravel the trace. In Figure 5, we present a sample portion of a trace of the execution of a TCP/IP protocol obtained using the *insert* and *prtrace* utilities. The first field shows the time the event occurred (in milliseconds since a fixed point in the past). The second field is the time since the last event. The third field is the type of event, whether we are CALLING the function or RETurning from it. If we are returning, the number of the return statement that caused us to return is also printed. *Insert* numbers the return statements in the source code to provide easy cross reference. The next field is the function name itself. If the event is RET, the function name is followed by the elapsed time since the routine was called, adjusted for the time taken by the calls to CASPER.

Function level tracing in this way is simple, and CASPER with the two utilities, *insert* and *prtrace*, make it easy to obtain. These utilities, along with CASPER's ability to log both user level records and kernel level records onto the same trace file, have proven to be very useful for the analysis of network protocols whose layers often cross user/kernel boundaries.

6. Summary

We believe that CASPER fills an important gap in supporting timing and tracing services. The high level interface that CASPER provides for those interested in instrumenting the system (for either performance measurement or debugging) has already proven to save an enormous amount of development time. Its availability might even encourage some studies that otherwise might not be done.

```

3139558.39 0.00 CALL  tcpoput
3139558.53 0.14 CALL  tcp_top_put
3139558.91 0.38 RET   1 tcp_top_put  0.32
3139559.05 0.14 RET   1 tcpoput  0.48
3139559.43 0.38 CALL  tcposrv
3139559.55 0.12 CALL  tcp_output
3139559.78 0.23 CALL  tw_dupmsga
3139560.30 0.52 RET   1 tw_dupmsga  0.46
3139560.44 0.14 CALL  tw_send
3139561.23 0.79 CALL  in_cksum
3139563.26 2.03 RET   1 in_cksum  1.97
3139563.38 0.12 CALL  tcpsend
3139563.53 0.15 CALL  ipoput
3139563.91 0.38 RET   1 ipoput  0.32
3139563.99 0.08 RET   1 tcpsend  0.43
3139564.08 0.09 RET   1 tw_send  3.22
3139564.20 0.12 RET   1 tcp_output  3.99
3139564.33 0.13 RET   1 tcposrv  4.12

```

Figure 5. Sample Trace of TCP/IP Protocol

Our choice to package CASPER as a device driver has given us unforeseen benefits. In addition to ease of installation, it has led us to develop a means of allowing user level programs to share the same timing and tracing facilities available to the kernel. In retrospect, this is ideal because in many situations, such as network protocols, the distinction between kernel and user levels is not a semantically natural one.

REFERENCES

1. Maurice J. Bach, *Design of the UNIX Operating System*, Prentice Hall, Englewood Cliffs, NJ, 1986
2. P.J. Weinberger, "Cheap Dynamic Instruction Counting", *AT&T Technical Journal*, Vol. 63, No. 8 Part 2, October 1984, pp. 1815-26
3. *UNIX User's Reference Manual*, AT&T Select Code 307-232, 1986
4. Robert Hagmann and Robert Fabry, "Program Page Reference Patterns", *Performance Evaluation Review*, Vol. 11 No. 4, p. 68
5. Stephen Tolopka, "An Event Trace Monitor for the VAX 11/780", *Performance Evaluation Review*, Vol. 10 No. 3, p. 121

NIDX - A REAL-TIME INTRUSION DETECTION EXPERT SYSTEM

*David S. Bauer
Michael E. Koblenz
Bell Communications Research*

ABSTRACT

The design of knowledge-based prototype Network Intrusion Detection Expert System (NIDX) for the UNIX® System V environment is described. NIDX combines knowledge describing the target system, history profiles of users' past activities, and intrusion detection heuristics forming a knowledge-based system capable of detecting specific violations that occur on the target system. Intrusions are detected by classifying user activity from a real-time audit trail of UNIX system calls, then determining, using system-specific knowledge and heuristics about typical intrusions and attack techniques, whether or not the activity is an intrusion. This paper describes the NIDX knowledge base, UNIX system audit trail mechanism, history profiles and demonstrates the knowledge-based intrusion detection process.

1. Introduction

This paper describes the design of the Network Intrusion Detection Expert System (NIDX) - a software system currently being prototyped by Bellcore as part of our effort to address network security issues. NIDX is a knowledge-based system that will examine a real-time, detailed audit trail of user activity on a computer system with the purpose of detecting security violations. We consider intrusion detection, or real-time audit trail analysis, to be one component of a more comprehensive security strategy. Other components, which are essentially preventive in nature will not be described here. These include data encryption, distributed authentication, secure operating systems, physical security and, most importantly, the definition and enforcement of personnel security policies.

The computer system being monitored by NIDX is known as the *target system*. NIDX is being designed to combine knowledge describing the target system and security related heuristics to form a knowledge-based system capable of detecting specific violations that are potential threats to the target system. Initially, NIDX is being developed to monitor the UNIX® System V environment, but its concepts may later be applied to other operating systems.

The next section reviews previous work and introduces the knowledge-based approach to intrusion detection. Our knowledge-based model applied to the UNIX System V environment is presented in Section 3. Sections 4-6 then describe the NIDX architecture, design, and the intrusion detection process. The concluding section points to some issues for further investigation.

2. Previous Work

Intrusion detection technology is based on the premise that, either in the attempt to break into a system, or once having broken into a system, an intruder will exhibit some form of abnormal behavior when attempting to access and control the system's resources. This abnormal behavior can then be detected by real-time analysis of an activity audit trail, allowing appropriate alerting and recommendations for action to the system security administrator. This basic idea has been previously expressed by Denning, et al, [1, 2] who defined an Intrusion Detection Expert System (IDES) model. The IDES model is a general purpose, system-independent model whose major components are:

† UNIX is a registered trademark of AT&T

- Subjects - entities that initiate some action, e.g. user and system programs.
- Objects - entities acted upon by subjects, e.g. files, memory locations, and physical devices.
- Audit Records - logs of actions taken or attempted by subjects on objects.
- Profiles - structures that express the observed normal relationships between subjects and objects, e.g. the access frequency for a particular user and a particular file.
- Anomaly Records - generated when a subject's action on an object is abnormal with respect to the corresponding profile.
- Activity Rules - describe actions to be taken when some condition is satisfied, e.g. update a profile based on new observed behavior in the audit records, or send a report to the security administrator based on analysis of anomaly records.

In the IDES model, intrusions are detected using the following approach: first, initialize and update profiles based on observed behavior of normal users, then monitor events on the system ("actions") looking for anomalous behavior. Anomalous behavior is discovered by comparing a users' current actions to their profiled history and determining by statistical methods that a person logged in as user *u* is exhibiting behavior not consistent with *u*'s usual behavior. For example, if a user performs actions (e.g. file removals) the number of which deviates substantially from the mean of that activity as shown by his/her profile, then such behavior is deemed anomalous. Once having detected such behavior, the system takes some appropriate action which may involve a report to the administrator.

The IDES model and subsequent prototype [3] were primarily designed to detect masquerade violations. A masquerade violation is where an authorized or unauthorized person uses a computer system under the identification of another user. The process described above can be used to detect masqueraders. However, it may not be able to detect other types of intrusions. For example, in the UNIX environment typical intrusions are unauthorized browsing, assumption of root permissions, and software and data theft. The above method may not detect these intrusions for the following reasons. First, users can slowly modify their behavior so that browse and theft violations appear to be normal behavior. Also, many intrusions occur quickly, i.e. a single command or two is entered to steal a piece of software or to exploit a commonly known or newly found security hole to gain root permissions. A very short sequence of anomalous behavior may not exceed the defined thresholds and, thus, not be detected.

In the next section we describe an alternative approach that addresses these issues using a *knowledge-based, system-dependent* model designed to characterize and detect particular intrusions.

3. NIDX Intrusion Detection Model

The NIDX model is based on the IDES system-independent model but is extended to include system-dependent knowledge to allow detection of specific security violations from the target system audit trail. An example of system-dependent knowledge is a description of the target system's file system and heuristic- and policy-based rules for detecting violations. Although the current NIDX model is being developed for the UNIX system, many of its concepts should be applicable to other operating systems. The components of the UNIX system model are:

- Users - sometimes called subjects. In the NIDX model users are ultimately held responsible for security violations. Thus, all activities are associated with a user. A user may not refer to an actual person, e.g. muucp.
- Objects - entities acted upon by users, e.g. files, devices, message queues, and directories.
- Audit Records - the detailed trace of successful and unsuccessful attempts by users to access system objects.
- User Profiles - maintain a history of users' interaction with the system. Profiles reflect each user's relationship to objects on the system, for example, user *x* to objects owned by another user *y*.

- **Anomaly Records** - generated when user activity suggests a suspicious event has occurred.
- **Intrusions** - the security violations that the UNIX system intrusion detection system detects. For example, unauthorized browsing, modification, or information theft.
- **Suspicious Events** - events that individually are not intrusions, but that may be an indication of an impending security violation. For example, a user who logs in late at night who has no history of such activity.
- **Knowledge Base** - contains knowledge about the system being monitored and heuristics for detecting intrusions.
- **Rules** - used to update session profiles from audit records, detect suspicious events from audit records, and detect activities which are indications of intrusions.

The goal of the NIDX model is to support the development of a system that can characterize and detect specific intrusions from an audit trail of user activity. In this approach intrusions are detected via a two step process. First, the audit records of user interaction with system objects are analyzed and placed in one of the following activity categories: browse, modify (includes create and write), delete, or copy (a combination of read and write.) Then, the activity is authorized according to policies and heuristics contained in the knowledge base. Thus, in order to detect unauthorized browsing, a system based on the NIDX model first detects that a user is browsing, then determines whether or not the browse activity is an intrusion.

Detecting browse requires NIDX to know the semantics of the audit trail records. For example, on the UNIX system where system calls are audited, this would require NIDX to consider the difference between *open-for-reading* and *open-for-writing*. Also, determining whether a detected activity is an intrusion requires NIDX to have access to policies and heuristics describing valid and/or invalid user behavior. An example of a policy is "users should not read other users files;" an example of a heuristic is "if some members of group *x* read file *y*, then other members of that group may read the file also."

The NIDX model described above supports this concept of intrusion detection. This differs from the IDES approach where normal user behavior is characterized and intrusions are hypothesized when a user's current behavior differs from their established norm. We believe the current NIDX model is appropriate for the UNIX system and other environments where formal and informal policies exist to allow differentiation between authorized and unauthorized behavior. To adapt NIDX to domains where there are few or no policies to govern user activities, the addition of techniques to distinguish users from one another may be necessary.

4. NIDX Architecture

Figure 1 depicts the NIDX architecture and data flow. Referring to the diagram: users interact with the target system, running programs, editing files, etc. An audit trail is transmitted in real-time to a UNIX workstation where NIDX is running. The NIDX system is physically co-located with the target system and communication is done via a physically secure point-to-point RS-232 line ensuring that the audit trail cannot be tampered with during transmission.

The knowledge base, user profiles, and rules are implemented in an expert system shell on the NIDX workstation. Once on the workstation, the system call trace is entered into the expert system's working memory where they are analyzed. When an intrusion is detected, an alarm showing the violation and an explanation of the reasoning that led to its discovery are displayed on the security administrators workstation. Then, the security administrator can instruct the target system to contain the intruder in some way. Initial containment mechanisms under consideration include restricting the intruder to a "safe" directory or logging him/her off the target system and disabling the suspected account. Another possible action is to limit the destructive damage the intruder can do while continuing to monitor with the goal of learning new attack techniques that can subsequently be included in the rule base.

A potential enhancement is to allow NIDX to automatically provide containment instructions to the target system. However, this should only be done at the discretion of the security officer and then only when intrusions are detected with a very high confidence level.

5. NIDX Design

The components of NIDX consist of the following: the set of intrusions and suspicious events to be detected, the knowledge base, audit mechanism, and user profiles.

5.1 Intrusions

This section describes the set of violations that NIDX is being designed to detect. These were selected because they represent the typical violations that can occur on most general purpose UNIX systems. Other violations, such as covert channels which are meaningful only on multi-level secure systems and denial-of-service which may be addressed as an enhancement to NIDX, are not part of the initial design. The intrusions may be committed by external (i.e. system crackers) or internal (i.e. authorized users abusing privileges) users.

5.1.1 Attempted External Break-in

An intruder may try to gain access to a system by attempting various login/password pairs until successful. Although the main purpose of NIDX is to detect intruders once they have successfully gained access, monitoring for attempted break-in is relatively easy and provides advanced warning of possible attacks on the system.

5.1.2 Browsing

Unauthorized browsing is the act of reading information that the browser has no legitimate right or permission to read. This intrusion is probably the most frequent offense for both internal and external intruders. But, it may be the hardest to distinguish from normal activity since most UNIX users engage in authorized browsing regularly. To detect unauthorized browsing, NIDX monitors users for reading in unusual, confidential, or restricted areas of the system.

5.1.3 Modification and Destruction

Direct modification and deletion of objects can be detected by intrusion detection systems and even by system administrators much sooner than browsing. However, other indirect forms of this intrusion may be more difficult to detect. For example, an intruder may insert a Trojan Horse [4] into a commonly used program. When that program is run by an authorized user, the Trojan Horse performs some covert activity that may result in the modification and/or destruction of objects. A Trojan Horse may be detected when a program or directory is modified during implantation. Also, the covert activity run from a Trojan Horse may be detected by the creation of set-user-id programs, the copying of files, or the modification of file or directory permissions.

5.1.4 Theft

Information theft over communication lines to external systems is a common goal of intruders who successfully break into a computer system. Further, this particular intrusion may be initiated by internal users. Popular objects of theft include tools, games, the operating system source, and proprietary software. One definition of theft is the unauthorized copy of information. Thus, it may be possible to detect theft by monitoring for copying. Unauthorized copying can be characterized by reading in system or restricted directories with concurrent writing to terminal lines, removable medium devices, or personal directories.

5.2 Suspicious Events

Suppose user *x* who has never logged in at night before logs in late one night. This activity is not a security violation per se — the user may have valid work to do. However, because systems are frequently broken into during off hours, such an activity may be considered a suspicious event if user *x* has no history of logging in at night.

Suspicious events may be used in conjunction with other information to detect violations. For example, suppose NIDX determines that user *x* is browsing public directories. Under normal circumstances such activity may be accepted. However, if one or more suspicious events have occurred, then NIDX may conclude that an unauthorized user is masquerading as *x*. This section describes the initial set of suspicious events that NIDX will monitor.

5.2.1 Abnormal Login Events

If a user logs into the system at an unusual time, terminal or network port.

5.2.2 Successful Internal Masquerade

A user who assumes the identity of another user should not be considered an intruder solely for that action. Indeed, the UNIX system *set-user-id* feature was designed to allow a user to enable other users to assume their identity when running certain programs. However, in that case there should always be a direct mapping between *set-user-id* program run and objects accessed. Any discrepancies may be signs of an intrusion.

5.2.3 Change in Work Area

If the set of directories a user works in remains relatively constant over time, then deviations in work area may be a sign of an intruder. If a user changes work area (i.e. accesses files in a new directory) then a suspicious event may be logged. The choice of the new work area (public or private) and the length of time since the last work area change are used to determine whether or not a suspicious event is recorded.

5.2.4 Awkward Use of the Computer

An external intruder may not be familiar with the system he/she has broken into. He/she may attempt to run commands that do not exist, try to change directories to those that do not exist, try to read files that do not exist, search directories looking for familiar commands and information, and extensively use the *man(1)* and/or system help commands. On UNIX systems, awkwardness is amplified by the many versions available.

A profile to measure the awkwardness of each user is maintained. If a user makes a number of mistakes during a session exceeding a predetermined threshold, a suspicious event may be logged.

5.3 Knowledge Base

The NIXD model includes a knowledge base describing the target system. The knowledge base is composed of information describing the target system's file system, users, and policies and heuristics for detecting intrusions. Most of the knowledge base contains information applicable to all or most UNIX systems. For example, the fact that the object */dev/rmt0* is a tape drive. Additionally, the knowledge base is designed to allow the specification of information pertaining to just one or a class of similar UNIX systems, such as those dedicated to a particular application. For example, the specification of policies relating to the use of database files for a locally developed application.

5.3.1 File System and User Knowledge

Knowledge about the target system's file system and users is essential for quick, accurate intrusion detection. For example, consider the */tmp* and */bin* directories: both directories can be read by all, however, whereas */tmp* can be written by all, */bin* can only be written by the system accounts *root* and *bin*. Thus, for an intrusion detection system, it is more important to notice a user writing into */bin* than into */tmp*. Without this knowledge, both directories must be treated equally.

The following categories were created to allow objects to be classified:

<i>read-public:</i>	read by all is permitted
<i>write-public:</i>	read/write by all is permitted
<i>exec-public:</i>	run/search by all is permitted
<i>read-restricted:</i>	read is a violation except listed users and programs
<i>write-restricted:</i>	write is a violation except listed users and programs
<i>exec-restricted:</i>	run/search is a violation except listed users and programs

Not all objects on a UNIX system will be classified. First, only those objects that have consistent access attributes with respect to all users (except *root*) should be classified. For example, */tmp* is open to all users and is classified appropriately. However, the objects in */tmp* may not be open and thus are not classified. Users' home directories are also not classified because they are open only to their owners and are restricted with respect to all other users. Finally, when entire directory hierarchies need to be classified, only the top

level directory must be listed. Unclassified files (and subdirectories, recursively) inherit, if specified, the classification of their parent. For example, `/bin` is classified as `exec-public`,¹ thus `/bin/date` is inferred to be `exec-public` as well. Note that this classification is more restrictive than the actual permissions on `/bin` which usually permit reading in addition to execution. However, since normal users only need to run programs in `/bin`, this classification helps browse and copy-related intrusions to be detected.

Objects unique to a specific UNIX system can be added locally if finer-grained classification is desired. This local knowledge is entered and maintained by the security officer or system administrator and can be as detailed as desired for each system being monitored.

Information about users such as their login names, home directories, user and group ids is maintained in the `/etc/passwd` file. Note that the presence and format of `/etc/passwd` is UNIX specific knowledge, but its contents are machine specific (it contains user information about a particular UNIX system.)

5.3.2 Intrusion Detection Policies and Heuristics

The final component of the knowledge base is a set of policies and heuristics for detecting and authorizing activities. The policies reflect formal and informal rules governing unauthorized user activities. The heuristics describe user activity that may potentially be an intrusion. Some examples are:

1. Users should not read objects in other users' personal directories. (policy)
2. Users should very rarely write other users' files. (policy)
3. Users do not generally open disk devices directly. (policy)
4. Users do not make copies of system programs. (policy)
5. Users who log in after hours often access the same files they used recently. (heuristic)
6. Users do not generally log in more than once simultaneously to the same system. (heuristic)
7. Writable directories are good places to put Trojan Horses. (heuristic)

These policies and heuristics have been gleaned from a variety of sources including security literature [5-8], discussions with security personnel and, in the future, from new attack techniques learned from experience.

5.4 UNIX System Audit Mechanism

5.4.1 System Call Trace

NIDX requires a detailed trace of users' activities and the objects they access. The standard UNIX system V accounting package logs every program executed, recording such information as program name, amount of disk I/O, and CPU time. However, it does not provide a record of the objects read or written.

In order to meet the objectives of the NIDX model it was found that the UNIX system kernel needed to be modified to provide a trace of system calls invoked by user processes. This was necessary because only at the system call level can user interaction with system objects be traced in real-time. Not all system calls need be audited: only those that manage objects and affect user privileges are traced. NIDX will initially trace the following system calls:²

1. On UNIX systems, execute permission on a directory allow it to be searched.
2. Readers familiar with UNIX System V release 3.1 will notice that system calls dealing with message queues, semaphores, shared memory, and the Transport Interface (TI) have been omitted. Auditing these system calls is necessary for complete intrusion detection, but, because the techniques needed to monitor them are similar to the listed system calls, they are not described here.

access	dup	link	setgid	stat
chdir	exec	open	setpgp	unlink
creat	fork	pipe	semid	

A typical audit record will contain, using the `open(2)` system call as an example, real and effective user/group id, process id, object, action, time stamp and success indicator. Thus, invocation of the system call:

```
open ("/usr/dave/x", O_RDONLY);
```

will result in the following trace record:

user:	195	call:	open	object:	/usr/dave/x
group:	130	flag:	read	object-owner:	195
euser:	195	errno:	0	object-group:	130
egroup:	130	time:	12345	type:	file

We modified the UNIX kernel to obtain a detailed trace of user access to objects. A preferred alternative is to use an equivalent trace in a standard format from a commercially available version of UNIX System V so that local kernel modifications would not be necessary. The level of auditing required for a C2 and higher rating as described in the Trusted Computer System Evaluation Criteria [9] and implemented in [10] appears to be sufficient for intrusion detection. As UNIX systems rated C2 and above with standard audit trail formats become available, we plan to investigate the applicability of the NIDX model to them.

5.4.2 System Program Trace

In addition to monitoring user/object interaction, certain events on the target system are important to NIDX. First, NIDX must know when a user logs in so that monitoring can begin. Also, NIDX must know when a user logs out so that monitoring can end after all processes executed on behalf of that user terminate. The `login` program will be modified to record each successful login as well as unsuccessful login attempts to allow attempted break-ins to be detected. The end of a session can be determined when a `getty` is restarted on a terminal line. The `getty` program will be modified to record such events.

Finally, a user can legally assume the identity of another user without logging-in again via `su(1)` and `set-user-id` programs. The objects accessible via a properly developed set-user-id program should be limited by the program itself. However, since `su` provides the user with a shell, object accessibility is unlimited. Thus, `su` will be modified to inform NIDX of successful attempts. Unsuccessful attempts are not logged since a user attempting to crack passwords can access the encrypted password from `/etc/passwd` at will.³

System programs log trace messages via a new system call implemented to allow non-kernel resident functions to contribute to the audit trail. The system call is protected so that only certain programs, currently `su`, `login`, and `getty`, can successfully invoke it to log messages. This prevents false messages from being logged by users attempting to subvert the intrusion detection process.

5.5 Profiles

Profiles are used to characterize and summarize the behavior of each user and are used to augment the intrusion detection process when knowledge of past activity would facilitate the analysis of a suspected violation. There are two groups of profiles: first, permanent profiles for each user that are updated at the end of each login session and second, separate session profiles maintained while the user is logged-in to distinguish current from past activities. This section describes the initial set of NIDX user profiles. Because profiles are computationally expensive to use and have significant storage requirements, part of the on-going research is to determine which profiles are necessary for effective intrusion detection.

3. Actually, once an intruder cracked a password he/she could merely log-in to the victim's account. However, if the root password is cracked, the intruder must generally use `su` since System V prevents root logins from devices other than the console.

5.5.1 Successful External Login Profile

Login profiles are organized primarily by time of day because, generally, most users have a consistent work schedule. There are login profiles for various time periods including week-day, evening, late-night, weekend-day, and weekend night. Each profile also lists the access method, e.g. hard-wired line or dialup. This information provides a consistent record of each users' general computer access habits.

5.5.2 Successful Internal Login Profile

The only way for a user to "login-in" as another user once on the system is via the `su(1)` command, given that the user does not already have super-user permissions.

A profile is maintained for each user and the logins that user successfully su's to. The profile includes user id, new user id and the work area directory profiles used while under the new id. This profile is necessary to distinguish activities done through a set-user-id program versus those done as a result of successful su.

5.5.3 Attempted External Login Profile

Intruders may use trial and error in choosing user ids and passwords when attempting to break into a system. Potentially useful user ids often tried are *guest* and *sys*. Tracking the user ids entered, number of attempts, device and time of day is useful in detecting patterns of attempted break-ins.

5.5.4 File and Directory Access Profiles

The file and directory access profiles maintained for each user describe that user's relationship to the files and directories owned by others. For example, the profile *dave-read-mike* is updated whenever user dave reads a file or directory owned by the user mike. Since there are too many objects on a system to monitor each individually, objects are grouped into classes and each class is monitored rather than individual objects. Part of the ongoing research is to determine the appropriate granularity of the object groupings.

The following sections describe the actual profiles. The '*' below denotes the actions performed on the objects. The actions monitored are: read, write, execute, search, and delete. The read profile is updated whenever a file is opened for reading. The write profile is updated whenever a file or directory is opened for writing. The execute profile is updated whenever a program is invoked. The search profile is updated whenever a directory is opened for reading. The delete profile is updated whenever a directory or file is removed. Each profile includes the following information: user id, profile type, e.g. *user-read-other*, event counters and time stamps.

5.5.4.1 User-*-other These profiles monitor one users interaction with another user's objects. Note that "another user" does not include system objects, which are monitored separately. Profiles can be created which monitor a users interaction with one other user's objects, e.g. *user-*-xyz*, or which monitor a users interaction with several user's objects, e.g. *user-*-abc,hij*.

5.5.4.2 User-*-self These profiles monitor a users actions to objects he/she owns. These profiles may be useful in detecting masquerade violations.

5.5.4.3 User-*-system These profiles monitor the use of uncommonly used or restricted system objects that, if accessed by a user, may result in a security violation. Examples are devices in */dev*, *uucp* files in */usr/lib/uucp*, and the system source.

5.5.5 User Id and Group Id Profiles

All *set-user/group-id* programs on a particular system should be known to the system administrator and NIXD because each is a potential source or target for security violations. Except for a few system commands, such as `su(1)`, `passwd(1)`, and `mail(1)`, *set-user/group-id* programs are rare and should only be used to provide access to files not available without special permission. Thus, each non-system *set-user/group-id* program will be profiled independently for all users. That is, this profile set covers the *set-user/group-id* programs that each individual user invokes. Deviations in the files or directories accessed or programs run from a *set-user/group-id* program may be an indication of a Trojan Horse or security flaw in the program.

5.5.6 WorkArea Profiles

A user's work areas are the directories he/she accesses during a login session. One way to focus the attention of the intrusion detection system is to notice that a user is not working in a previously established work area.

A work area profile is an ordered list of all the directories a person accesses over a given time with those directories most recently accessed at the front of the list. Although a user's primary work area may change over time as his/her tasks and skills change, deviations allow the intrusion detector to focus on activities occurring in the new directories. Thus, this profile is not used as the sole basis for detecting an intruder, rather it is used to detect suspicious events.

5.5.7 Awkwardness Profile

This profile monitors the awkward events described in section 5.2.4.

6. The Intrusion Detection Process

This section presents two examples of techniques developed for NIDX. The first demonstrates the intrusion detection process - how the knowledge base and heuristics are used to distinguish authorized from unauthorized behavior. The second shows how NIDX determines that a particular activity, in this case copying, has occurred from the detailed audit trail.

6.1 Detecting Unauthorized Browse

Detecting unauthorized browse is accomplished in two steps: first, browse activity is detected from the audit trail and second, authorization of the activity is determined. Consider the following user command:

```
$ cat /usr/mike/x
```

Let us assume this command was entered by some other user, say user dave, and that the directory /usr/mike is the home directory of another user, mike. The system call trace (abbreviated) is:

```
user: 195      call: open      object: /usr/mike/x      type: file      flag: read
```

Detection of browse is relatively easy since browse activity results from opening files and directories for reading. Section 6.2 shows how browse is distinguished from other activity also involving reading, e.g., copying. The following represents part of the browse authorization process:

1. If the object is */etc/passwd* or another object with a classification of *read-public* in the knowledge base, then read is authorized.
2. If the object is classified as *read-restricted* and if user 195 or the program reading the object is not listed as having permission to read the object, then an alarm is generated.
3. If no explicit classification exists for the given user and object, then heuristic authorization is performed as described in the following list items:
4. If the object is owned by the user who accesses it and no suspicious events have occurred, then read is authorized.
5. If the object is owned by the user and one or more suspicious events have occurred and the object is not a directory listed in the user's recent work-area-profile, then an anomaly record is generated. Several anomaly records are needed before an alarm is generated.
6. If the object is in the directory hierarchy of another user, say user *x*, and is not in a "generally public readable" directory, e.g., *~x/bin*⁴ or *~x/rje*, then an anomaly record is generated because of the policy "users should not read other user's files." Specification of "generally public readable" directories is

4. The symbol *x* is an abbreviation for the home directory of user *x*.

contained in the knowledge base.

7. The previous heuristic can be augmented with the use of profiles. For example, authorization may be granted if user 195 or members of user 195's group had shown previous history of reading that particular object.

This example has shown how simple intrusions can be detected from just one audit trail record. The list contains some of the heuristics used to determine authorization of browse or read activity when simple binary decisions cannot be made. Part of the ongoing research is to generalize and combine heuristics.

6.2 Detecting Copy Intrusions

Distinguishing copy intrusions from browse intrusions is important because once a file is copied, the copier has unlimited access to it. Also, since devices are actually special files on UNIX systems, the command:

```
$ cp /usr/mike/x /dev/diskette1
```

copies the file to floppy disk which may result in information theft. Copy intrusions are detected the same way as browse intrusions: first the copy is detected, then authorization is determined. Detection of copy activity is difficult because several audit records are needed to infer that copying has occurred and there are many methods to copy a file each resulting in a potentially different audit trail. Authorization can be done using knowledge of user/object relationships and heuristics in the same manner as browse intrusions.

Consider the following three examples. Each shows a different command to copy the file /usr/mike/x to /usr/dave/y and its respective audit trail (abbreviated).

Example 1

```
$ cp /usr/mike/x /usr/dave/y
```

user: 195	call: exec	object: /bin/cat		
user: 195	call: open	object: /usr/dave/y	type: file	flag: write
user: 195	call: open	object: /usr/mike/x	type: file	flag: read

Example 2

```
$ cat < /usr/mike/x > /usr/dave/y
```

user: 195	call: open	object: /usr/mike/x	type: file	flag: read
user: 195	call: open	object: /usr/dave/y	type: file	flag: write
user: 195	call: fork			
user: 195	call: dup	object: stdin		
user: 195	call: dup	object: stdout		
user: 195	call: exec	object: /bin/cat		

Example 3

```
$ cat /usr/mike/x | cat > /usr/dave/y
```

user: 195	call: pipe			
user: 195	call: open	object: /usr/dave/y	type: file	flag: write
user: 195	call: fork			
user: 195	call: dup	object: stdin		
user: 195	call: dup	object: stdout		
user: 195	call: exec	object: /bin/cat		
user: 195	call: fork			
user: 195	call: dup	object: stdout		
user: 195	call: exec	object: /bin/cat		
user: 195	call: open	object: /usr/mike/x	type: file	flag: read

Although the audit trail for each command is different, some re-occurring themes can be exploited. Each command eventually opens */usr/mike/x* for reading and */usr/dave/y* for writing. Also, the audit trail patterns *fork-dup-exec* and *pipe-fork-dup-exec* are indications of I/O redirection.

This knowledge is used in the following procedure to determine that a file was "copied": Audit records are processed using forward-chaining strategy with the goal of discovering either a "restricted"⁵ file opened for reading or an "unrestricted" file opened for writing. If that goal is reached, the goal state *detected-copy* is set and a subgoal is created to substantiate it. Satisfying the subgoal depends on whether opening a restricted or unrestricted file satisfied the *detected-copy* goal state. That is, whether a "restricted" file was opened for reading first or whether an "unrestricted" file was opened for writing first. If a "restricted" file was opened first, then the subgoal is satisfied if an "unrestricted" file is opened for writing next. Conversely, if the *detected-copy* goal state was reached when an unrestricted file was opened for writing, then a restricted file must opened for reading to satisfy the subgoal. Satisfying the subgoal is done using a backward-chaining strategy.

If the process exits before the subgoal is reached or another violation is detected, then the *detected-copy* goal and the subgoal are removed from consideration. If patterns for I/O redirection are detected in the audit trail, meaning that the source of all I/O has not yet been determined, then analysis must continue. Once the "copy" is detected, then authorization is done using the techniques described in the last section.

7. Conclusion

Conventional security tools provide various levels of protection against potential intruders. However, this protection is given only as long as the preventive mechanisms are not compromised. Once these mechanisms are defeated by a sufficiently determined intruder, the underlying system is again left open to tampering. Moreover, this tampering is often untraceable by conventional means. Thompson, for example, demonstrates in [11] how a clever system programmer can insert devices into system software that are virtually undetectable by current security mechanisms.

A complementary mechanism to intrusion prevention is detection. Conventionally, this is achieved by providing detailed audit data that allows security officers to detect intrusions, assess damage and then repair security holes. The approach described in this paper in which the audit trail is analyzed in real-time using expert system technology is a substantial improvement over batch mode audit trail inspection. A key objective of real-time analysis is that it may allow real-time containment of an intruder's activity, thereby limiting the damage that can be done.

We have described the architecture and design of the NIDX expert system for network intrusion detection. The NIDX system, specifically targeted for the UNIX System V environment, is being designed to detect

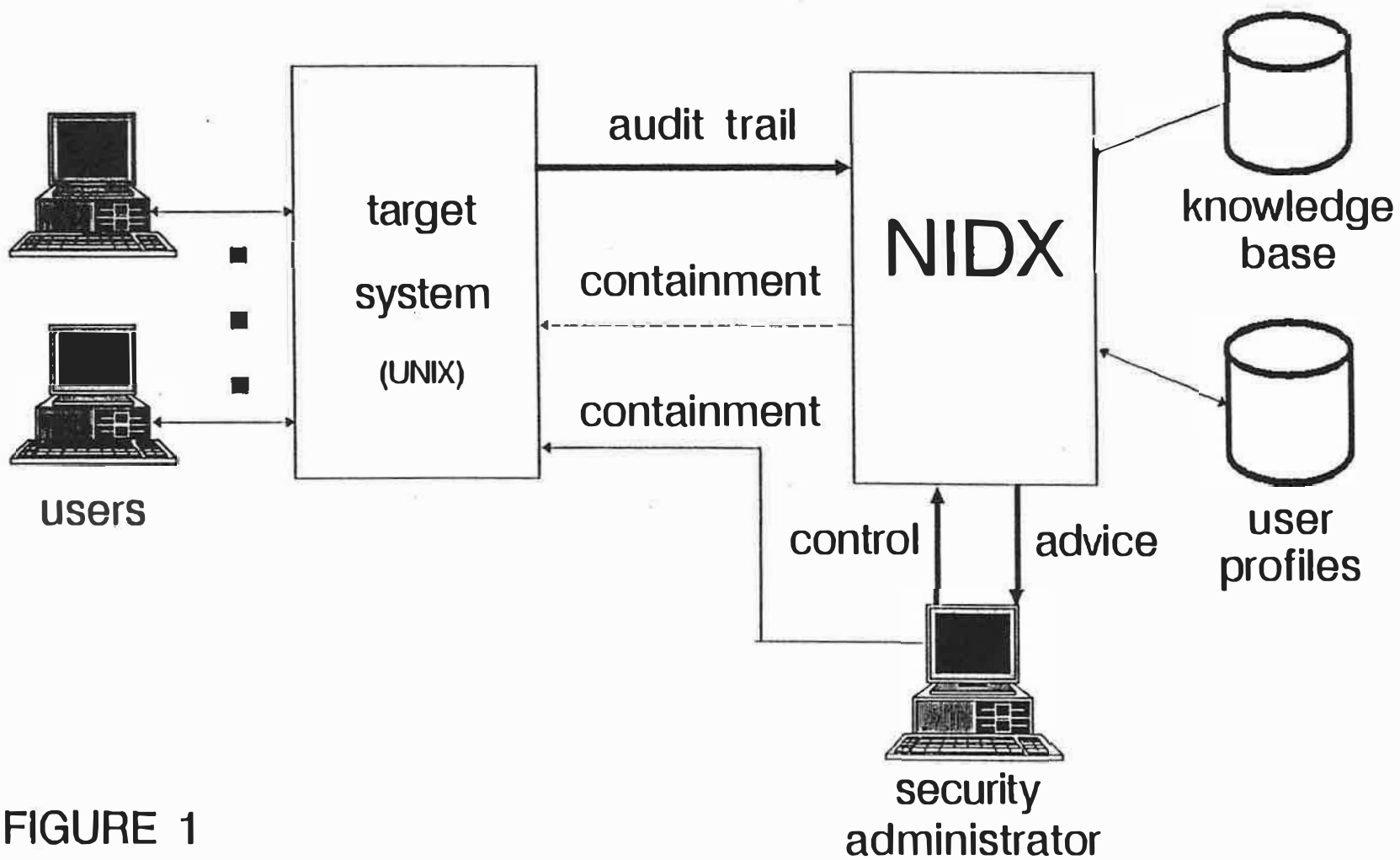
5. With respect to the user generating the audit trail.

security violations in real-time by analyzing a trace of UNIX system calls using system-specific knowledge, history profiles of users' activities, and heuristics about typical intrusions and attack techniques.

There are also several open issues we are investigating. Since it may be difficult for a general purpose computing system to provide sufficient resources to its users while monitoring itself for intruders, we are investigating the role of parallel processing in intrusion detection where additional processors are used to offload the auditing and monitoring functions from the target system. In addition, we are investigating the tradeoffs between fast detection and the number of false alarms generated, using data from external prototype field studies to determine an acceptable balance. Finally, the prototype environment is expected to prove a base for refinement. As new attack techniques are discovered, the appropriate defensive mechanisms will be incorporated into the system.

REFERENCES

- [1] D. E. Denning and P. G. Neumann, *Requirements and Model for IDIES - A Real-time Intrusion-Detection Expert System*, SRI Project 6169, SRI International, Menlo Park, CA, August 1985.
- [2] D. E. Denning, D. E. Edwards, R. Jagannathan, T. F. Lunt, and P. D. Neumann, *A Prototype IDIES: A Real-Time Intrusion-Detection Expert System*, Final Report, SRI Project ECU 7508, SRI International, Menlo Park, CA, August 1987.
- [3] T. F. Lunt and R. Jagannathan, *A Prototype Real-Time Intrusion-Detection Expert System*, Proceedings of the 1988 IEEE Symposium on Security and Privacy, April 1988.
- [4] C. Stoll, *What Do You Feed A Trojan Horse?*, Proceedings of the 10th National Computer Security Conference, September 1987, pp 231.
- [5] J. Lobel, *Foiling the System Breakers*, McGraw-Hill Book Company, 1986.
- [6] R. Farrow, *Security For Superusers*, UNIX/WORLD, May 1986, pp 65.
- [7] P. H. Wood and S. G. Kochan, *UNIX System Security*, Hayden Books, 1985.
- [8] R. Farrow, *What Price System Security?*, UNIX/WORLD, June 1987, pp 54.
- [9] Department of Defense Computer Security Center, *Trusted Computer System Evaluation Criteria*, DOD-5200.28-STD.
- [10] V. D. Gligor, E. L. Burch, C. S. Chandrasekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W. D. Jiang, G. L. Luckenbaugh, and N. Vasudevan, *On the Design and the Implementation of Secure Xenix Workstations*, Proceedings of the 1986 IEEE Symposium on Security and Privacy, April 1987, pp 102.
- [11] K. Thompson, *Reflections on Trusting Trust*, ACM Turing Award Lecture, Communications of the ACM, August 1984.



Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses[†]

Nick Lai

Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024
lai@cs.ucla.edu

Lapsley, Lai, Hayes, and Schoch
P. O. Box 21
Berkeley, California 94701
lai@llhs.com

Terence E. Gray

Bridge Communications Division
3Com Corporation
2081 Stierlin Road
Mountain View, California 94043
bridge2!teg@sun.com

ABSTRACT

Conventional Discretionary Access Control mechanisms, even in systems with Trusted Computing Bases, are vulnerable to Trojan Horse and Computer Virus attacks because a process typically acquires all of the access privileges of the user invoking it. An important security goal is to limit a process to only those accesses intended by the user. This paper presents a mechanism which attempts to satisfy this goal, using a program's run time arguments as a hint of the user's intentions.

1. Introduction

Protecting the information in computer systems from illicit exposure, modification, and destruction is a difficult problem, one that is of critical importance in both military and commercial environments. This information is vulnerable to a daunting variety of assaults: software booby-traps, network eavesdropping and packet forging, abuse of superuser privilege, compromise of backup tapes, and confidence tricks that can be played upon users, operators, and administrators alike, to name a few.

Of all the ways a system's security may be compromised, the most popular among system security breakers seems to be the *Trojan Horse* attack. A software Trojan Horse is a program which 1) appears to offer some desirable service, 2) has some hidden functionality, 3) is designed with malicious intent, and 4) improperly uses properly granted access rights to achieve its goals [2]. An example of a Trojan Horse might be a "game program" left on a public bulletin board, which, in addition to being a potentially entertaining game, occasionally sets bits randomly on the users' disk drives in order to maliciously destroy data. A classic UNIX[‡] Trojan Horse scheme involves taking advantage of naive command search paths that include publicly writable directories. The malicious individual puts a program with the same name as a popular utility (e.g. "mail") into one of these directories. A user who has this directory early in his search path may inadvertently execute the Trojan Horse instead of the real utility. The malicious program, which upon execution is imbued with all of the victim's access rights and privileges, may do anything to the victim's files that their rightful owner could. Upon completing its subversive tasks, the program will then execute the *real* "mail" program so that the victim does not suspect that anything untoward has happened.

[†] This research was supported in part by the NSF Coordinated Experimental Research Program under grant NSF/MCS 8121696 and by the IBM Corporation under contract D850915.

[‡] UNIX is a trademark of AT&T Bell Laboratories.

A particularly dangerous form of Trojan Horse is the *Computer Virus*. A Computer Virus is "... a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows." [1]

A Computer Virus may additionally carry malicious code which is triggered once the Virus has gained appropriately broad access privileges.

Both of these types of assaults exploit a weakness in current Discretionary Access Control (DAC) implementations in which processes are accorded all of the privileges that their invoking users possess [4,8]. A DAC is a "means of restricting access to objects based on the identities of the subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access authorization may, at its discretion, be capable of passing (perhaps indirectly) that access type or a subset on to any subject." [4]

Let us examine this problem algebraically, in a notation similar to that presented in [9]. First we will make the following definitions:

U is the set of all users; u is an element of U .

S is the set of all subjects (processes); s is an element of S .

O is the set of all objects (files, devices, programs); o is an element of O .

A is the access set of all types of operations (read, write, append, execute, etc.).

In a typical DAC system, the set of all access triples is

$$M = (U \times O \times A)$$

U and S are interchangeable here, since no distinction is made between the privileges of the process and the privileges of the user.

The accesses allowed for user u is

$$M_u = (u \times O \times A)$$

where M_u is constrained by any Mandatory Access Controls that might be in effect.

Now let us assume that user u wants to run program o . The user makes the request to run program o , and M_u is checked to see if u has execute permission on the program object o . If the permission exists, then program o is run on behalf of user u in a new-created process s (or by the pre-existing process s if the operating system in question does not create a new process for each program run by the user).

Under conventional DACs, the accesses available to s are

$$M_s = (u \times O \times A) = M_u$$

In contrast, the accesses that the user intended for s , M_{I_s} , is some subset of the totality of access triples available to the user. That is,

$$M_{I_s} \subset M_u$$

Trojan Horse and Computer Virus programs exploit the fact that they are not limited by the intent of the user. That is, the access triples M_E , exploited by the malicious program are again a subset of the totality of access triplets available, but are distinct from the accesses intended by the user:

$$M_E \subset M_u$$

$$M_E \neq M_{I_s}$$

There is no single solution to this problem. However, we assert that by applying the "Least Privilege Principle" [8] to processes' file accesses, the damage that can be done by programs that contain Trojan Horses and Computer Viruses can be significantly limited. Therefore, while traditional DACs are designed such that

$$M_s = (u \times O \times A) = M_u$$

we propose a system in which

$$M_s = M_{I_s} \subset M_u$$

That is, a system in which the access triples available to a process are limited to those intended by the user.

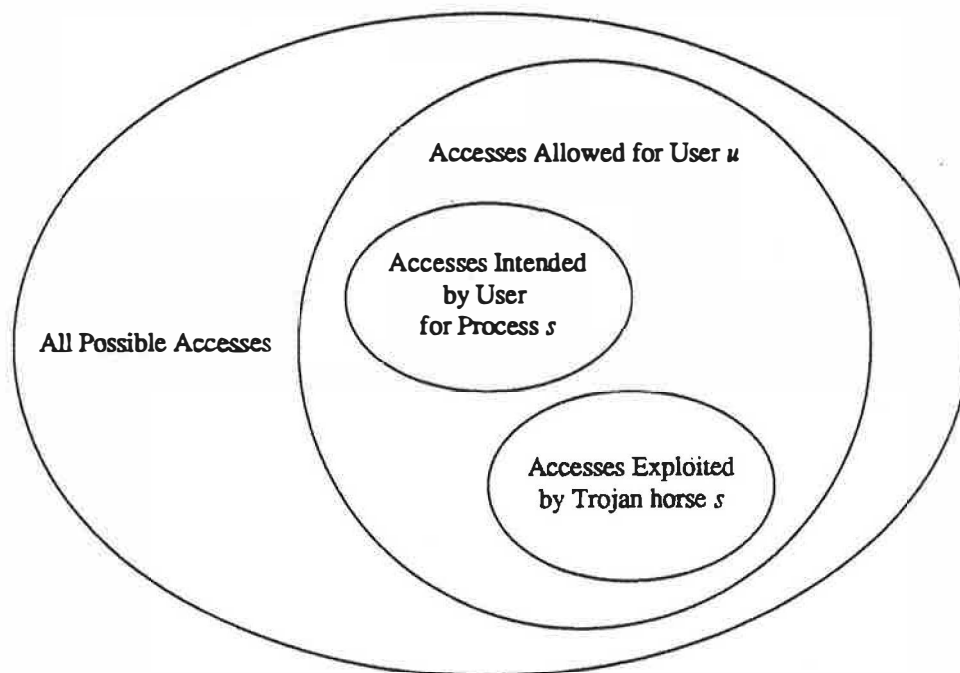


Figure 1. Access privileges in typical DACs.

Figure 1 illustrates the problem with typical DACs. The largest ellipse represents the set M of all access triples for all users. The circle labeled "Accesses Allowed for User u " represents M_u . The two smallest ellipses represent M_{I_s} and M_{E_s} .

We have set out to restrict a process to only those accesses that are both *permitted* under the Mandatory and Discretionary Access Controls in effect (M_u), and *intended* by the user of the program (M_{I_s}). The goal is to prevent permissible but unintended accesses which might be attempted by a Trojan Horse.

There are a number of ways that this goal may be achieved. One possible scheme might involve having the security kernel interactively ask a user to authorize every operation on every object which a process running on the user's behalf might initiate. At the opposite extreme, one might design an expert system which would automatically determine if a requested operation is in violation of the *intent* of the user invoking the program. Neither of these schemes is feasible, the former due to the intolerable amount of user intervention required, and the latter because of the shortcomings of current technology. Clearly, a system which combines some user intervention with a degree of machine inference of user intent must be built if the goal is to be satisfied in a practical manner.

In this paper, the authors present just such a mechanism. The key idea is to use a program's run time argument list as a statement of the user's intent, and to flag or restrict access to resources not mentioned in the argument list. The system asks the user for authorization for the small subset of accesses which might not be resolved automatically by the system. This mechanism will limit the damage caused by Trojan

Horse attacks to only those objects passed to these malicious programs as arguments. While this scheme will severely limit the impact of discretionary attacks, it is not a total solution and is intended to be used in conjunction with other protective measures such as those described in [6,7].

2. Scope of this Research

The protective mechanism described in this paper is designed solely to prevent a program from accessing (for read, write, append, truncate, execute, creation, deletion, locking, or access parameter change) those files that are not passed to it as arguments. Exceptions to this rule are 1) globally readable files, which may be read even if they are not included on the command line, and 2), a special set of rules governing the creation, use, and disposal of "temporary files". This scheme offers no protection to those files which are passed to a program as arguments, either from integrity violations (modification) or disclosure via covert channels. However, private files not included on the argument list are protected from both disclosure and integrity violations since all surreptitious accesses to these files are prevented. Moreover, disclosure of information in a file on the argument list via creation of a new file will be detected.

It is true that this mechanism can be subverted if certain critical data structures in the kernel can be modified by malicious individuals. However, the threats that this scheme addresses are present even in systems which are built upon a Trusted Computing Base (TCB) [3]. Thus, an examination of the issues related to protecting kernel data structures is not within the scope of this paper.

In addition to providing the above described protection, the ideal system should impose little overhead in CPU cycles and memory usage on the host operating system, and should not require undue amounts of user intervention. Finally, the mechanism should be binary compatible to its unmodified host operating system. A mechanism which satisfies all of these requirements is presented below.

3. Solution Strategy

3.1. Brief Overview

At the highest level, the mechanism can be viewed as follows: users invoke commands from *trusted* command interpreter processes, and unless special provisions have been made for the particular application by the system manager, the resulting process is considered to be *untrusted*.¹ A trusted process in this context is one that can make use of any and all access triples available to the user, that is, within M_u , and further, can invoke other programs to operate upon the objects within this constraint. In contrast, untrusted processes are *not* free to use any access triple in M_u ; each time an untrusted process requests a file system operation, the kernel scans a data structure called the Valid Access List (VAL). The VAL consists of the arguments with which the program was called plus the names of any new files that the process or its progeny have created. The VAL also contains a reference count of how many processes are sharing it. If the file upon which the operation was requested is not on the VAL, the user is asked whether or not he wishes to allow the access (the mechanism for securely querying the user is discussed in § 5.1).

3.2. File Accesses

The distinction between trusted and untrusted processes, together with the VAL, are the elements out of which this security mechanism is built. Our concepts of trusted and untrusted processes give us a small class of processes to trust implicitly, and a much larger class of processes whose actions we must suspect and regulate. Our VALs give us criteria upon which to judge a process' requests for access to objects. These tools, and the following five rules, make up the kernel of the mechanism:

- 1) Whenever a file access is requested by an untrusted process, the VAL is searched for the requested file.
- 2) If the file is on the VAL, the request is performed using the user's access privileges.

¹ This is true for systems like UNIX in which processes spawn processes, but does not apply to systems like MULTICS. A similar mechanism can, however, be applied to systems where the command interpreter and invoked commands are effectively the same process.

- 3) If the file is *not* on the VAL and the operation requested is an open for read, and the file in question is globally readable, then the operation succeeds.
- 4) If the request is for creation of a file, the operation is performed using the user's access privileges, with the protections on the file set such that only the user has any privileges to manipulate the new file. Further, the file is entered onto the VAL, where it is tagged as being a New Non-Argument file (discussed in § 3.4).
- 5) If the operation requested does not fall under rules two through four, then the operating system asks the user to authorize the operation. If the user authorizes the request, then it is performed using the user's access privileges. Otherwise, the operation fails. In either event, the fact that this exceptional condition occurred is noted in the system log.

A more relaxed policy of only querying the user in the event of questionable modification operations is also possible. This would have the advantage of greatly reducing the number of queries that the user would have to answer in the course of normal activity. The disadvantage of this policy is that the illicit disclosure of data becomes easier to arrange. We shall call this weaker approach the Modified Policy.

3.3. Trusted and Untrusted Processes

The first principle of this scheme is that users are able to generate any command strings that they wish, and that these commands will be executed by the system within the confines of their access privileges. Therefore, the keyboard input from the users is to be accepted as properly reflecting user intent. This is achieved by designating the user's command interpreter task as a trusted process.

Users do not usually access files directly from their command interpreters. Instead, they invoke programs which display, edit, or otherwise process the files to which they have access. Unless specific administrative actions have been taken to the contrary, all programs which a user invokes will run as untrusted processes. When a trusted process directly spawns an untrusted process, a VAL is created and is associated with a new Untrusted Process Family (UPF) which initially is made up of that first untrusted process. Any processes later spawned by that first untrusted process, or its progeny, will be considered part of this UPF, and will all share the same VAL. This is to allow cooperating subprocesses to communicate information via shared temporary files. Note that the processes spawned by an untrusted process will always be untrusted.

Some programs are inherently promiscuous in their file accesses. For example, some programs access files recursively, that is, they are called with directories as arguments and then traverse the file system trees rooted at those directories, performing operations on some or all of the objects therein. These programs do not fit well into our scheme, since they always access files which are not on the VAL and would demand an intolerable amount of user intervention. Fortunately, few users write programs which are file system recursive, and it is probably safe to assume that the system utilities which perform this kind of service are free of malicious code, as long as malicious users do not gain superuser privileges and thus write access to these utilities. Therefore, part of this scheme is a mechanism by which a program can be marked as *trustable*. A program which is marked as trustable and which is invoked by a trusted process, will run as a trusted process, and will therefore have its file accesses constrained only by the privileges of the invoking user, without regard to the program's argument list. Of course, a trustable program which is invoked by an untrusted process will run as an untrusted process. This trust granting scheme is used to mark as trustable *only* those utilities which routinely access non-argument files, as well as the standard command interpreters. We claim that there are only a few dozen programs which require this special treatment (see § 5.3 for a list of the programs which would be marked as trustable under 4.3 BSD UNIX). The Modified Policy described in § 3.2 has the additional advantage of reducing the number of programs which have to be marked as being trustable in order to provide a reasonable level of user intervention.

Some users may write programs or command files which would be inconvenient to use unless they were run as trusted processes. We therefore need a mechanism by which a user can explicitly specify in the program invocation that a program should be run as a trusted process. The implementation of this mechanism is operating system dependent, and the reader should see § 5.3 for a discussion of its implementation under UNIX.

3.4. New Files

Many legitimate programs create new files during their execution that may not be named on the program's argument list. These may be either temporary or permanent files. Temporary files are those files that are used to hold the results of various stages of a computation, such as the data produced by the stages of a compiler, which are deleted once the computation is completed. Permanent new files are not deleted upon the completion of a program. An example of a permanent new file might be the finished executable that is the output of a compiler. New files which are not named on the argument list, be they temporary or permanent, are necessary resources which must be available to even untrusted processes. Since these files do not fall neatly under our scheme we must develop a set of special rules for dealing with them.

Since untrusted processes must be free to create new files that are not named on the argument list, file creation is allowed, within the constraints of the user's access privileges. New files are entered onto the VAL so that they can be shared among cooperating members of a multi-process computation. As these new files are entered onto the VAL, they are specially marked as being New Non-Argument (NNA) files.

NNA files do have restrictions on how they can be manipulated. They are created with their protection attributes set such that only their owner may access them; thus preventing read or modify access by independent processes invoked by other users. The process which created the new file, and the other processes in its UPF, may not change these permissions, as this would provide an easy way for a Trojan Horse to disclose the contents of an argument file. When an NNA file is removed by a member of the UPF, its entry on the VAL is deleted.

While this allows us to have temporary files, and to create permanent files which are not on the argument list, it does create some security problems. For example, a Trojan Horse could create temporary files whose names are made up of the data in the argument files which can be gathered by a cooperating malicious program. This is an example of a covert storage channel. Another malicious program could create hundreds of files and directories with random names wherever the user has the permission to do so. We therefore notify the user of any NNA files which are still extant upon the termination of a UPF. Even with this notification, it might take the victim quite a while to clean up the mess. We acknowledge both of these problems. This first problem is a disclosure problem that, as stated at the beginning of this paper, is not within the scope of our solution. The second problem is not integrity critical, but is potentially annoying. Fortunately, it would be easy to eliminate this annoyance. Upon the termination of a process which has created NNA files, in addition to simply listing all of the NNA files which are still in existence, we could ask the user if he would like to have all of these new files deleted.

4. Examples

Now that we have presented the mechanism, we will examine how it applies to various kinds of operations.

4.1. The C Compiler

The 4.3 BSD UNIX C compiler is made up of 6 parts: a pre-processor (`"/lib/cpp"`), the compiler (`"/lib/ccom"`), the optional optimizer (`"/lib/c2"`), the assembler (`"/bin/as"`), the loader (`"/bin/ld"`), and a master program (`"/bin/cc"`) which calls these various stages in the proper order and with the appropriate arguments. The pre-processor takes two arguments: the source file, which should already exist, and an output file, which should not exist and which is created by the pre-processor. The compiler, the optimizer, and the assembler all take two temporary files as arguments, the first contains the output of the previous stage, and the second is created by the current stage. The loader loads together a number of files, some of which may be temporary, others which may already exist, and still others which are globally readable. The loader stores its output in another file, which may or may not already exist. The temporary files which are employed in the compilation process are deleted by the master program upon termination of the procedure.

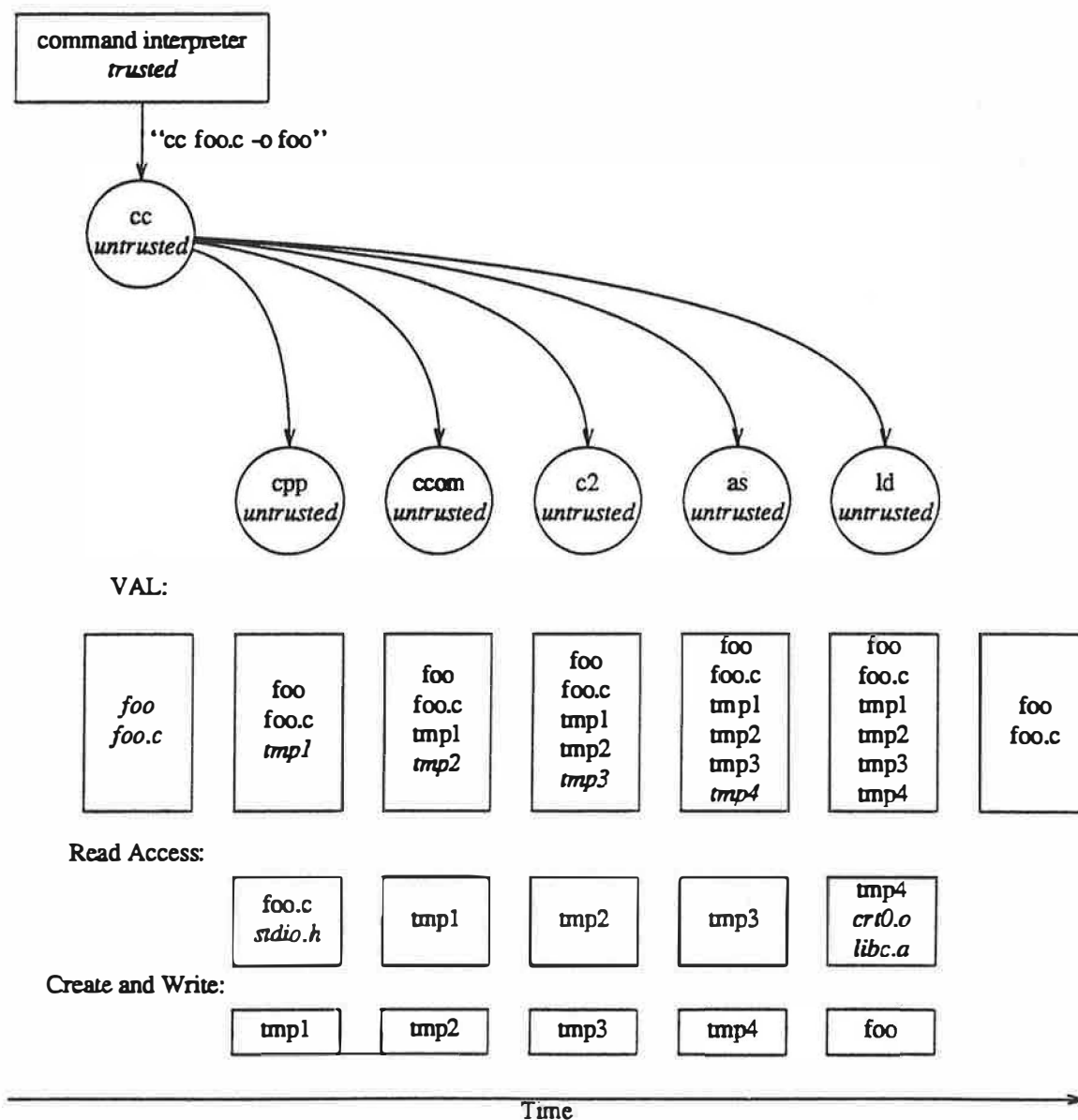


Figure 2. The operation of the C compiler under the proposed scheme.

Figure two illustrates how the compilation process proceeds, and how the VAL evolves during its execution (assuming that the C compiler has not been marked as being trustable). Italicized names in the "VAL" boxes are the names of files that have been added to the VAL at that particular stage of the operation. Italicized names in the "Read Access" boxes are the names of globally readable files which are read at that stage of the operation. Let us suppose that the user has entered the command "cc foo.c -o foo," which asks that the C source code in file "foo.c" be compiled and that the resulting executable be placed in "foo." The master program, "/bin/cc," is called from the trusted command interpreter, at which point a VAL is set up and initialized with the command line arguments (for the sake of readability, we have left off the flag "-o" from the illustration, although it is present in the VAL, since we don't presume to reliably distinguish flags from file names). The master program ("/bin/cc") calls the pre-processor ("/lib/cpp") with the arguments "foo.c" and "tmp1"². The pre-processor reads "foo.c," which is

² In reality, the C compiler temporary files are named things like "/tmp/crm086813," but we have used "tmp1," "tmp2," etc., for simplicity's sake.

allowed since it is on the VAL. Let us suppose that "foo.c" includes "/usr/include/stdio.h," the globally readable definition file for the C formatted-I/O library. Since this file is globally readable, the pre-processor is able to read it and include its text into the pre-processed version of "foo.c." The pre-processor creates the file "tmp1." The file "tmp1" is *not* on the VAL, since the process ("/bin/cc") which invoked the pre-processor command "/lib/cpp foo.c tmp1" is untrusted. However, since "tmp1" does not exist, it is allowed to be created and is entered onto the VAL as an NNA file. The pre-processor writes its output into "tmp1" and exits.

The master program then calls the compiler ("/lib/ccom") with the arguments "tmp1" and "tmp2." The compiler reads its input from "tmp1," which is allowed since it is on the VAL (albeit marked NNA), and creates the file "tmp2," a new temporary file which is again entered onto the VAL as being an NNA file. The optimizer ("/lib/c2") and assembler ("/bin/as") stages are identical in their file behavior to the compiler ("/lib/ccom"), except that they operate upon different temporary files.

The loader is called from the master program using the following invocation: "/bin/ld /lib/crt0.o tmp4 -lc -o foo." "/lib/crt0.o" contains the standard start-up code for all C programs. With this is loaded the object code for "foo" which is in "tmp4," and the standard C library (the "-lc" argument instructs the loader to find a library named "libc.a" in one of several standard system directories, typically "/lib"). "tmp4" is accessible by the loader because it is on the VAL as an NNA file. "/lib/crt0.o" and "/lib/libc.a" are accessible by the loader because they are globally readable, and the loader is only performing read operations on them. Note that globally readable files not supplied as arguments are not placed on the VAL, but read access is still permitted on these objects in accordance with rule 3 in § 3.2.

The output of the loader is stored in "foo." Note that since "foo" is on the VAL and is *not* an NNA file, any access which the loader wishes to perform on the file is allowed (within the access privileges of the user), and further, when the program terminates, the kernel will not notify the user even if "foo" was created by one of the untrusted processes. The procedure is now complete, so the master program deletes all of the temporary files ("tmp1" through "tmp4") and exits. The kernel searches the VAL and finds that there are no NNA files left, and the process is allowed to terminate quietly.

4.2. Trojan Horse

As an example of how this system foils Trojan Horse and Computer Virus attacks, let us examine the "naive search path" Trojan Horse presented in § 1. Suppose that the user invokes the command "mail," and that his command interpreter, because of an unwise search path, first finds a program called "mail" in "/usr/public" (a globally writeable directory on many UNIX systems). M_u is examined to see if u has permission to access the object in question. Assuming that the malicious individual has left the Trojan Horse globally executable, a process s is created to run the program on the victim's behalf. Since the program is by default untrusted, the created process is untrusted. Since the user did not give any arguments to the program, an empty VAL is created for the new UPF. At this point, M_s is equal to the set of access triples reflecting the files on the VAL with whatever access rights the user has to those files, plus all of the globally readable files for read, plus the ability to create new files under rule 4 in § 3.2.

Now suppose the intent of the Trojan Horse is to make a copy of the user's homework assignment, "/a/classes/cs162/u/hw1." The program attempts to open this file for reading. The system performs rule 1 of § 3.2 by scanning the VAL for the file name, and does not find it. It then tries rule 3 by checking to see if the file exists, which it does, and then checking to see if it is globally readable, which it is not. Therefore, by rule 5, the user is queried as to whether he wishes to allow this access. The (even moderately informed) user declines. The open request therefore returns an error value, and the wily Trojan Horse "cleans up" by executing the *real* "mail" program and removing itself. The event is automatically logged by the kernel, so even if the user does not actively inform someone, a system operator will find a record of the event when he next reads the log.

Note that even under the Modified Policy this Trojan Horse would be prevented from copying the homework to a new file or appending it to some already existing globally writeable file.

5. Implementation Issues

While the above material has presented the new system in some detail, there are a number of issues which remain to be addressed. For example, the mechanism for querying the user in a secure manner is a critical, albeit operating system dependent, function that must be provided. Creating and maintaining the VAL is a performance critical operation, and its implementation is therefore important. Another implementation detail is the mechanism by which users are allowed to explicitly specify in their command invocations that a given program is to be run as a trusted process. Besides implementation details, there is also the question of user education: no system can operate well if its users are ignorant of the benefits and pitfalls that it can provide. A security mechanism is enhanced greatly when its users are aware of it at least peripherally, and know what to expect. This section will deal with all four of these issues.

Note that many of the implementation details discussed below are UNIX specific. However, the basic concepts of the mechanism should be largely portable to different operating systems.

5.1. Querying the User

Our security mechanism is worthless if a malicious program can masquerade as the user and give its own answers to queries directed at said user. The system needs to be able to print a query and collect the user's answer while preventing any processes from writing to or reading from the terminal.

As noted above, this problem is highly operating system dependent. The authors will here present their mechanism for doing this in a 4.3 BSD UNIX system.

Preventing output from appearing on the screen is a simple task. All output destined for a terminal must be entered onto the appropriate output *clist* (character list) in the kernel. Characters are put onto the *clist* by writing to the file which corresponds to the particular terminal. By adding a semaphore to the state of each output *clist*, the kernel can cause all programs to block on writes to the terminal until such time as the query has been completed and the kernel releases the semaphore. When the semaphore is set, only the kernel may put characters onto the output *clist*; namely, for writing the query, and for echoing the user's responding keystrokes. Further, when the semaphore is set, any other program which wishes to query the user must wait on the semaphore.

Characters from the terminal are put onto the input *clist*. Unfortunately, programs (with the appropriate access to the terminal file) can also push characters onto the input *clist*. However, since characters from the keyboard are put onto the input *clist* by special interrupt or polling routines, semaphores can be put onto the input *clists*, too. When the semaphore is set, only the terminal driver routines may put characters onto the locked input *clist*. In this state, the kernel will read from the input *clist* up to a newline, and treat that input as the user's response to the query.

Note that if the process in question is not associated with a terminal, or the controlling trusted process has terminated, then the query automatically fails and the requested operation is denied.

The above mechanism is complete: it totally negates the possibility of a rogue program faking a user's response or otherwise interfering with the query process. It is left it as an exercise for the reader to determine how to implement this mechanism on other systems.

5.2. Implementing the VAL

The VAL is the center of activity in this security scheme. It must therefore be implemented in such a way that the creation and maintenance of this critical data structure is as efficient as possible. However, this is again a machine and operating system dependent problem. And again, we will discuss how to implement it under 4.3 BSD UNIX.

VALs are created whenever a trusted process creates an untrusted process. That is, when a trusted process *exec*'s an untrustable program. It is therefore in *exec* that the VAL must be created. Fortunately, *exec* already builds up a buffer which contains all of the arguments which are to be passed to the new process. Instead of releasing this buffer when the *exec* call is done, we set a pointer in the "proc" structure to point to a new structure, the VAL, which contains a pointer to the argument buffer and a reference count, which is initially set to one. During the life of the UPF, the reference count on the VAL will be incremented as processes are spawned by the original untrusted process and its progeny, and will be decremented as they terminate. Files will be added and deleted from the VAL as temporary files are

created and removed. When the last member of the UPF is done, and *exit* decrements the reference count to zero, the memory associated with the now-defunct VAL will be freed.

Scanning the VAL must be fast. Since the buffer is just a list of ASCII file names (and other arguments), a simple comparison with the questionable file and the argument list will suffice. Most programs which take multiple files as arguments access those files in the order that they appear on the argument list. Therefore, instead of always searching the VAL from the beginning, we maintain a "current pointer" into the VAL. This pointer points to the last file name we matched on, and this is where we will start the next VAL search.

On a DEC* VAX, we can make use of the *matchc* instruction, which searches a string for a substring, to speed up VAL searches even more.

5.3. Trusted Programs

In § 3.3 we asserted that the number of programs that would need to be marked as trustable is relatively small. We have identified the following thirty-two programs in the 4.3 BSD UNIX system as requiring trustable status:

Program(s)	Reason
/etc/getty and /bin/login.	Marked trustable so that the user's login shell will be a trusted process. ³
/bin/ls, /usr/bin/find, /bin/du, /bin/tar, /bin/tar, /etc/dump, /etc/restore, /bin/rcp, /bin/diff, /bin/chown, and /bin/chgrp.	These programs need to perform recursive file accesses.
/bin/csh and /bin/sh.	These are the standard UNIX command interpreters.
/usr/ucb/script	Runs a command interpreter and saves all terminal output into a file.
/usr/ucb/vi, /bin/make, /lib/cpp, /bin/mail, /usr/ucb/dbx, and /usr/ucb/mail.	These often access non-argument files.
/usr/lib/sendmail, /etc/ftpd, /etc/rshd, /etc/rlogind, /etc/telnetd, /etc/fingerd, /etc/tftpd, /etc/talkd, /etc/ntalkd, and /etc/uucpd.	These network daemons need to access various files in response to network requests.

Table 1. Trustable programs in 4.3 BSD UNIX.

All told, these trustable programs represent only eight or nine percent of the total number of programs in the vanilla 4.3 BSD UNIX release. The italicized programs are those that would not require trustable status if only modify operations are being intercepted by the mechanism.

A program is marked as being trustable by setting a particular bit in its inode structure. Since all of the bits in *i_mode* are significant in most UNIX flavors, a bit from *i_spare* must be used for this purpose.

5.4. Invoking Arbitrary Programs as Trusted Processes

As noted in § 3.3, a mechanism to allow the user to specify on the command line that a program should be run as a trusted process is a desirable feature.

The basic rule is this: A trusted root (superuser) process can, via the new system call *trust*, set a bit in its process state such that when it next *exec*'s, the program which overlays it will run as a trusted process.

* DEC and VAX are trademarks of Digital Equipment Corporation.

³ The kernel must create the *init* process with trusted status.

In order to explicitly run a program which is not marked as trustable as a trusted process, we posit a program "trust," which is *setuid* root and marked as trustable. This program checks to see if it is being called from a trusted process. If it is not, then it makes an entry to the system log and terminates. If it is, then it calls the *trust* system call, does a *setuid* to the user, and calls *execv* with the rest of the arguments on the command line.

The reason that the command interpreter itself can not be made to provide this function via a "trust" keyword is that these interpreters run with normal user privileges, and the *trust* system call is only available to superuser processes. Moreover, it is desirable not to have to modify each command interpreter to provide this function.

5.5. Educating the User

It would be ideal if a system could be built which restricted M_s to precisely M_I , with no user intervention and no CPU or memory overhead. However, we do not live in a perfect world, and we have to pay for what we get. This system was designed with certain assumptions about what kind of mix of these three overheads was acceptable. A small amount of user intervention is essential to make this system both broad in scope and realizable.

Having a security system interact with possibly naive users is a risky proposition. We believe, however, that even a pathologically naive user will not be able to subvert this mechanism in a major way. He may suffer disclosure or damage of all of his personal files if he blithely answers "yes" to all access queries. But the system operators on a secure system *must* check the log often, and abuses *will* be detected and remedied. Meanwhile, all of those users who behave in at least a marginally prudent manner will be protected from the ravages of most malicious programs.

Therefore, in order to maximize the protection afforded to the users of this system, users need to be provided with a minimum set of information with which to interpret system queries. This information could be provided by the kernel along with each query for those users that it can identify as being "new." Or this information can be given to each user when he is given his account. Regardless of the method of dissemination, information of the form: (1) "Trojan horses and Computer Viruses do exist, and they typically have the following behavior ..."; (2) "the security mechanism built into this operating system will exhibit the following behavior to the following kinds of malicious programs"; and (3), "here are some guidelines to follow in responding to system queries," must be provided to each user of the system.

6. Comparison with Other Work

In [5], Karger presents a mechanism which has a motivation similar to the one presented in this paper. His mechanism is built around a "table-driven file name translation mechanism that has knowledge of the normal patterns of use of a computer system." Under this scheme, for each program in the users' environment, some trusted administrator must specify the form of the *names* of the files that the program should access, under normal circumstances. For example, the Fortran compiler on a VMS system can be expected to read files which end in ".for" and write files which end in ".obj"; so, if Fortran is called on the file "xxx.for," reads from the file "xxx.for" and writes to the file "xxx.obj" will not be questioned. At run time, all file requests by a process are scrutinized to see if the names of the files upon which the operations are requested fit the description of valid file name patterns found in the table.

We find this mechanism to be cumbersome and incomplete in comparison to the system described in this paper. First of all, there is no security *by default* for any program which has not been explicitly evaluated and entered into the table. Indeed, Karger's paper does not talk about how to provide protection to new or local programs, other than to add them to said table. In our system, the addition of new programs into the system requires no new work on the part of the system administrator - they are *automatically* treated as un-trustworthy programs, and scrutinized as defined above. But most importantly, we think that it is *unrealistic* to rely solely upon the *pattern* of the file names. For one thing, it does not scale well to systems which do not employ extensions on all file names (for example, UNIX). In fact, how many programs, even in the VMS world in which this mechanism was developed, really have dependable behavior with respect to the *patterns* of file name extensions that they are going to access? In addition, the mechanism presented by Karger deals poorly with multiple-process jobs, while we believe that our

Untrusted Process Family scheme elegantly allows reasonable sharing of files while remaining faithful to the philosophy of the system: ease of use and automatic protection.

Because our mechanism provides protection by default to new or local programs, is dependent upon the arbitrary names of the files instead of the patterns they may form, does not require an administrative analysis and characterization of each and every program in the system, and reacts gracefully to multiple-process jobs, we conclude that our mechanism is more complete and flexible than that presented in [5].

7. Conclusions

Typical Discretionary Access Control (DAC) mechanisms are extremely vulnerable to Trojan Horse and Computer Virus attacks, because they accord to every process all of the access privileges possessed by their invoking user. The complete solution to this problem would be to limit a process' access privileges to only those *intended* by the user. Implementing such a solution in full measure is currently infeasible.

However, the authors have proposed a mechanism that approximates this ideal solution in a system that is effective and which imposes little overhead in terms of user intervention and machine resources. The system requires no special hardware, and should be binary compatible to the operating system to which it is grafted.

This goal is achieved by limiting a process' file accesses to only those files that are named in the invocation of the program. The mechanism also has a set of rules for dealing with new, temporary, and globally readable files.

The authors feel that a prudent approach to computer security is to continue to develop new mechanisms that, while not in themselves complete, are mutually independent and complementary. We believe that the approach described in this paper is one such mechanism.

REFERENCES

- [1] Cohen, Fred, "Computer Viruses," *Computers & Security*, vol. 6, no. 1, pp. 22-35, February 1987.
- [2] Denning, Dorothy E., *Cryptography and Data Security*, Addison-Wesley Publishing Company, Reading, MA, 1982.
- [3] *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense, National Computer Security Center, December 1985. DOD 5200.28-STD.
- [4] Downs, Deborah D., Jerzy R. Rub, Kenneth C. Kung, and Carole S. Jordan, "Issues in Discretionary Access Control," in *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, pp. 208-218, 1985.
- [5] Karger, Paul A., "Limiting the Damage Potential of Discretionary Trojan Horses," in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 32-37, 1987.
- [6] Pozzo, Maria M. and Terence E. Gray, "A Model for the Containment of Computer Viruses," in *Proceedings of the Second Aerospace Computer Security Conference*, pp. 11-18, Washington, DC, December 1986.
- [7] Pozzo, Maria M. and Terence E. Gray, "Managing Exposure to Potentially Malicious Programs," in *Proceedings of the Ninth National Computer Security Conference*, pp. 75-80, Gathersberg, MD, September 1986.
- [8] Saltzer, Jerome H. and Michael D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, September 1975.
- [9] Smith, Terry A., "User Definable Domains as a Mechanism for Implementing The Least Privilege Principle," in *Proceedings of the Ninth National Computer Security Conference*, pp. 143-147, Gathersberg, MD, September 1986.

Concurrent Access Licensing

S. Margaret Olson

Paul H. Levine

Stuart H. Jones

Stephanie Bodoff

Stephen C. Bertrand

Apollo Computer, Inc.

330 Billerica Road

Chelmsford, MA 01824

(617) 256-6600

...[yale,mit-eddie,decvax]!apollo!molson

ABSTRACT

Current software licensing techniques do not adequately address the needs of users in large distributed networks. Site licensing requires that a license be bought for every machine in the network, even if only a few users need the application. Node locking places a high administration burden on both the user and software vendor. Neither licensing mechanism is appropriate for large groups of users who need frequent, but not full time, access to an application.

The Network License Server (NLS) is a portable concurrent access licensing system built on Apollo's Network Computing System (NCS). The system manages licenses for all applications in the network, and dispenses application licenses as they are demanded. Licenses are returned to the server when the application exits. NLS consists of a server, an application library, and administration tools.

1. Introduction

People expect the price of services to be proportional to both the functionality provided and usage. Software on a large distributed network in many ways represents a service. Users on any node in the network can access and run the application, even if there is only one copy. To capture and charge for the usage of a software application, vendors have traditionally offered node locked licenses and site licenses. A node locked license ties the application to a specific machine with special hardware or software. The application will only run on that one machine. A site license is an installation-wide license; the number of users of the application is theoretically limited only by the number of users in the network.

Installations with only a few users of a given software application are satisfied with node locked licenses for that application because it is, in effect, priced proportionate to the amount of use. In environments where many users spend only a small fraction of their time using an application, the price of a dedicated node locked license may not be cost effective, so the site may not invest in the application. If the application is very useful, a site may buy one node locked license

and force users to share the one node on which the application will run. In either case, this means a reduction in revenue for the software supplier and a reduction in productivity for the end user.

Alternatively, if use of a particular application at a site is very widespread, the site may purchase a site license. This solves the user's problem of insufficient licenses, and it eliminates the need to move to one of a few licensed nodes. However, if the software is used only part time by many users, the site may be paying more for the software than is necessary. Since software vendors can not accurately gauge the number of users at a site, they may charge a premium for the site license.

Concurrent access licensing offers an alternative to site and node locked licensing. Rather than restricting the number of machines on which an application may run, concurrent access licensing restricts the number of people who may use the application at any one time. Several license servers, each responsible for a portion of the total number of licenses for a software application, are installed on strategically distributed nodes in the network. These license servers may administer licenses for any number of applications. When a user executes an application from any node in the network, the application locates a server that holds licenses for the application and requests a license from it.

Concurrent access licensing has several advantages. The application may run on any system that is networked to a license server that manages licenses for the desired application. Usage of the application depends on neither the location of the application nor the location of the user. The number of licenses for a given application is clearly defined by the software vendor; the customer can not increase the number except by purchasing more licenses from the vendor. Since each server holds only a portion of the licenses for a given application, some licenses are available even if a server node goes down. Each server can grant licenses for any number of applications simultaneously.

Built on Apollo's Network Computing System (NCS) [Dineen 87], NLS (Network License Server) has been developed to provide concurrent access licensing in large heterogeneous networks. In designing this system our goal was to provide a service that scaled to both large and small networks, and could support a wide range of software applications. Most very large networks consist of many types of machines running many types of system software, as well as many different applications. In a large network, users, administrators, and software developers want a uniform licensing interface across all machines. With a uniform licensing interface users and administrators need learn only one license management system, rather than one for each vendor. Software vendors preserve maximum uniformity between versions of the system targeted for different machines.

At this time we are aware of a few vendors who are developing proprietary concurrent access licensing systems. To our knowledge, these systems will be able to manage licenses for only the supplying vendor's software. There are application usage licensing systems for timesharing machines; these measure the amount of application usage on one machine. There are no references in the literature to general purpose distributed licensing systems.

2. Architecture

NLS is a portable system that consists of a server, an application library, and a set of administration tools. The server manages a database which contains the licenses and information about current license usage. The application library is provided to software vendors to be bound in with the licensed application. The library finds the license servers and manages the actual license

acquisition and release. The administration tools communicate with servers to provide users with the ability to update the database and to get information about current and historical usage of the licenses. The system is illustrated in figure 1. Individual components are described in greater detail in section 2.1. Section 2.2 discusses design issues.

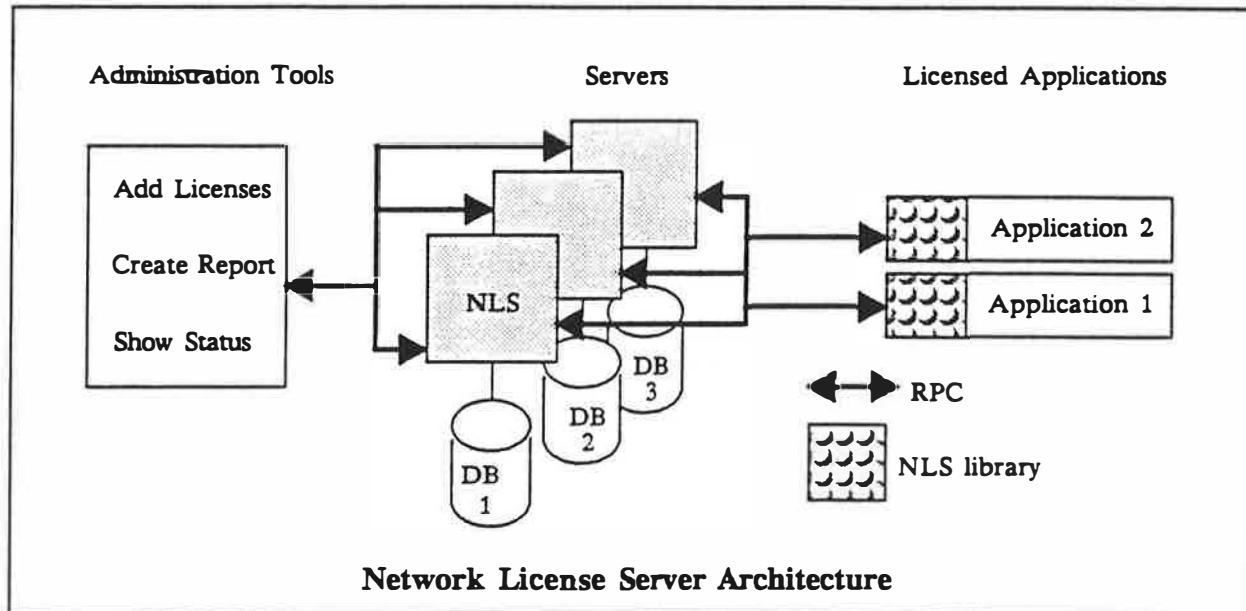


Figure 1

2.1 System Components

2.1.1 License Database

The database consists of two sections. The first section contains the licenses, which are of the general form (vendor, product, version, valid dates, number of licenses, number of licenses in use). Each vendor is identified by a UUID (universal unique identifier) which is created by the vendor with a tool provided by NCS. UUIDs are guaranteed to be unique across space and time, so there is no possibility of two vendors creating the same UUID. Products are identified by integers chosen to suit the convenience of the vendor. Product versions are represented by eight character strings. This creates the three tiered hierarchy illustrated in figure 2. The valid dates are the license start and end dates; the license is only available for use during this period. The number of licenses is the number of concurrent uses that this license record represents, and the number of licenses in use is the number currently being held by applications.

The second section of the database contains information about the current usage of licenses. Each usage record represents the licenses held by one instance of the application, and is of the form (vendor, license-id, number in use, transaction id, user id). The license-id identifies the license record from which these licenses were acquired. The number in use is the number of licenses being held by the application; applications may require more licenses for machines with more memory or other features which increase the usefulness (value) of the software service. The user id identifies the user running the application.

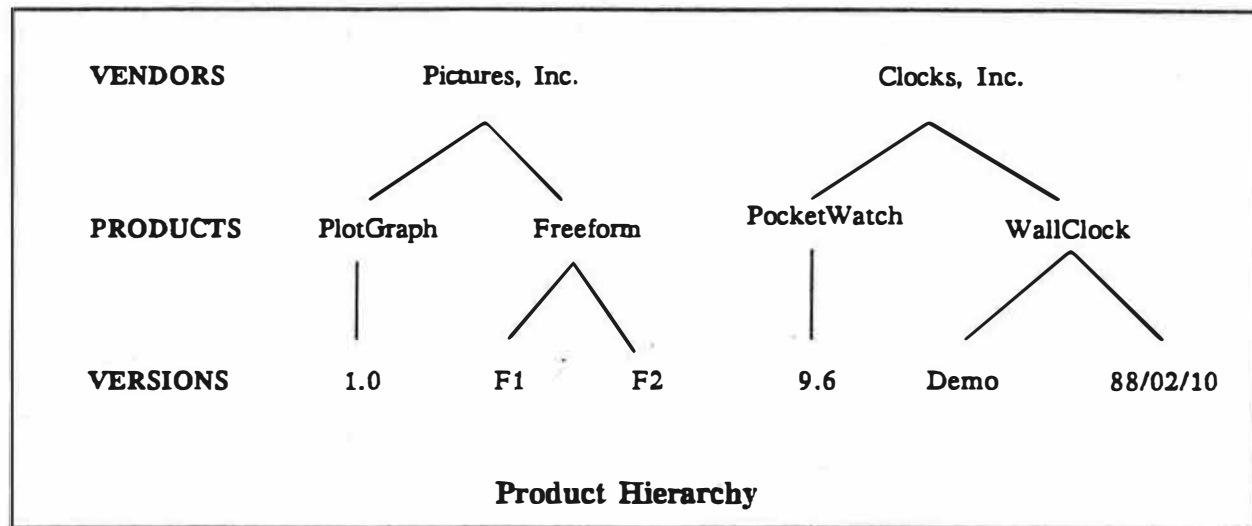


Figure 2

2.1.1.2 The Server

The NLS server manages the license database. The server communicates with its clients, the application and the administration tools, via remote procedure call (RPC) [Nelson 86]. The underlying RPC mechanism is provided by Apollo's Network Computing System. The server exports two interfaces, a set of operations for the application and a set of operations for the administration tools. Each operation is recorded in a log file that can be examined for license usage patterns by system administrators to determine if more licenses are needed for any particular application.

There are five application operations: *request*, *wait*, *wait-stat*, *release*, and *check*. *Request* requests the server for some number of licenses of the given (vendor, product, version) combination. If enough licenses are available the operation returns a transaction id to the application. This transaction id is used in subsequent *release* and *check* calls. *Wait* is like *request*, except that if a license is not available the server returns a transaction id and places the user on a queue. The application calls *wait-stat* every few minutes to see if a license has become available. Waiting users are placed in a queue; the first one on the queue gets the first license that becomes available. If a waiting application does not call *wait-stat* within two minutes the user is removed from the waiting queue. *Release* frees a license for use by another user; it removes a license usage record from the database. *Check* is a breath-of-life message from an application that tells the server that the application is still running and still holding its license.

There are seven administration operations: *add vendor*, *delete vendor*, *add license*, *delete license*, *delete vendor*, *show users*, and *dump log*. *Add vendor* adds the information that is common to all products of a vendor to the database, and associates a name with the vendor's UUID. *Add license* adds a license record for a given vendor to the database. *Delete vendor* and *delete license* delete those items. *Show users* shows all of the current users of a (vendor, product, version) tuple. This allows a user who has been denied a license, or who is waiting on a license, to find out who is holding the needed licenses. *Dump log* returns events logged in the log file.

2.1.3 The Application Library

The application binds with the NLS library. This library supports the application operations described in the previous section, and is responsible for making all of the actual RPC calls to the server. When the application makes an `nls_request` or `nls_wait` call to initiate a *request* or *wait* operation, the library contacts each of the appropriate servers until one of the following occurs: a license has been found, it has been determined that no licenses are available, or the user has been placed on every wait queue. Similarly, when the application calls `nls_wait_stat` the library makes a *wait-stat* call to every server at which the user is waiting. The application itself has no knowledge of the location or number of servers.

The library also supports the notion of a node locked license. There is a local text file that may contain license passwords associated with vendor UUIDs. Before initiating any RPC, the library checks for the existence of this file, and if a license exists does not request one from the server. If the library succeeds in obtaining a node locked license, it ignores check and release calls from the application. This allows the vendor to provide both node locked and concurrent access licenses without modifying the application. Users with node locked licenses do not need the NLS server or any of the administration tools.

2.1.4 The Administration Tools

There are four administration tools: create licenses (`nlspace`), edit database (`nlsadmin`), show status (`nlsstat`), and create report (`nlsrpt`). Create licenses is used by the vendor to make a password representing a license record. The password is entered by the user into the edit database tool, which makes the rpc call to add the license record to the database. Edit database is also used to add vendors. Show status shows the current usage of a (vendor, product, version) tuple; it can either discriminate by server or provide a global view. Create report uses the *dump log* operation to show the usage of licenses over a period of time. Both edit database and create report act on a per server basis.

2.2 Design Issues

2.2.1 Scalability

NLS was designed to scale to very large (thousands of machines) networks. As a computing community grows in size, the kinds and number of software applications used by that community also grows. Therefore the server must be able to accommodate large numbers of application transactions issued against a large number of licensed applications. Since there is a practical limit to the amount of service one process can provide, the system must scale to many servers as well as many clients.

Scaling up service to a large user community impacts the choice of underlying communications facility. One alternative is to use a virtual circuit facility to establish and maintain communications between the server and the requesting client for the entire time that the licensed application is in use. Using this form of long-lived active link addressed several major problems in detecting licenses orphaned by failed machines or network service interruptions. The server could detect a loss of contact with the application holding a license and initiate license recovery procedures. However, most systems place limits (based on available system resources) on the number of simultaneous virtual circuits. In these cases, the maximum number of licenses would be artificially restricted to the maximum number of allowed virtual circuits.

The NCS implementation of RPC is based on datagram service. It consumes system communications resources only while the server and client (application or administration tool) are actually communicating. There is no limit on the number of simultaneous client-server pairs.

2.2.2 Fault Handling

Occasionally an application will exit without releasing the licenses it is holding, or a server node may become separated from the application by a network partition. To detect either of these conditions the server and client must communicate with one another periodically. Once stranded licenses are detected the server can release these licenses for use by other instances of the application.

One way to approach this problem is to have the server query the application periodically, and if the application does not respond, to release the licenses. This is an appropriate strategy if the application is no longer running. But an application separated from the server by a network partition may continue unaware that its licenses have been released by the server. The server can not distinguish an application that is no longer running from an application that is stranded on the far side of a network partition.

Some vendors may wish to terminate the application in the event that a license is released by the server. To this end the application, when it requests a license, specifies an expiration period. The application sends a breath-of-life message to the server within this period to renew the license. At each license renewal the application has an opportunity to reset the reservation period. The ability to reset the renewal period prevents the application from having to make calls to the server during time-critical periods. If the license renewal call fails, the application knows that the server may release the license, although the server will in fact only turn in the license if another user requests it. The action the application takes when a renewal fails is left to the discretion of the application implementor; to date we have seen applications that terminate, applications that ignore the fact that the license disappeared, and applications that keep requesting a new license until either a new license is obtained or the user elects to exit the application.

2.2.3 Security

We had three primary security considerations in designing NLS. First, to guarantee that the licensed application actually contacts a valid license server and waits for a license to be granted. Second, to ensure that the licensing information originates from the application vendor and is installed without modification at its intended site. And lastly, that the license server system be no easier to break than other software licensing systems.

The best software based security measures, whether network based, node locked, or special hardware dependent, can be defeated by a determined criminal. The NLS protection scheme depends on an encryption key hidden in both the license server and the vendor operated license creation tools. This key is used to encrypt the database, a vendor specific password, and communications between the server and tools. A second key, selected by the vendor, is used to encrypt licenses and communications between the server and application.

The vendor key prevents users with access to the password creation tool from generating license passwords for products supplied by other vendors. The NLS encryption key is used to secure non-vendor specific information, and the vendor keys themselves. When applications communicate with the server, the NLS library includes a timestamp in the message. This timestamp is encrypted with the vendor key by the server, and returned to the library with the result of the remote procedure call. The library checks to make sure that the timestamp matches its own encrypted version of that timestamp. This prevents a software pirate from collecting the bits sent in response to various calls, and setting up a server that echoes certain responses.

DES [NBS 77] is an encryption scheme which has proven powerful enough to discourage all but the most determined software pirates. All of the encryption services are performed with the DES encryption algorithm.

2.2.4 Availability

The cornerstone of any network-based service is the availability of the server to requesting clients. Since nodes and networks occasionally fail, providing license server availability was a critical concern during the design of the license server. We looked at two options: replicating the database across multiple license servers, and partitioning the database among multiple servers. We considered both static and dynamic partitioning. With dynamic partitioning, licenses are automatically distributed across all servers when first added to any one server. Statically partitioned licenses are explicitly divided among servers by the system administrator.

At first glance the most appealing option is database replication, since it appears to offer the greatest amount of availability with the lowest administration cost. The license server updates the database every time a license is granted to an application, so weakly consistent replication is not sufficient: every server must know how many licenses have been granted by all servers. Transaction processing protocols, together with weighted voting schemes, provide strongly consistent replication that allows a majority of servers to continue if some become unavailable [Bernstein 87]. These protocols require that many messages be passed between servers, which in turn causes a performance degradation. In the event of a network partition license server availability is reduced; the minority of isolated servers will shut down. In addition, transactions processing protocols are difficult to implement.

Dynamic partitioning is simpler to implement and provides greater availability than a true replicated database. With dynamic partitioning, when licenses are added to any server, they are automatically divided evenly among all available servers. The partitioning is accomplished with an atomic protocol similar to a two phase commit protocol. This protocol also allows administrators to move licenses explicitly from one node to another. Except to distribute or redistribute licenses from one database to another, the servers do not communicate with one another, so all servers are available after a network partition.

A major problem with both dynamic partitioning and replication is that the licenses are not securely tied to any database, and databases are in the hands of the users. Banks with transaction processing systems to manage their databases keep these databases locked securely away from the account holders. Suppose that information in a license received from a vendor dictates in which machine a license is first installed. With either dynamic partitioning or replication the user could add licenses to this master node and set up a number of derivative databases. The user could then copy all of these databases to some other location, delete the originals, and start over. On the second pass the malicious user would use a different set of nodes for the derived databases. The user now restores the first set of databases. If the databases are replicated, the network must be partitioned between the first set and the second set. If the database was dynamically partitioned the user has no need to artificially partition the database. In either case the database is easily subverted.

We have looked at a number methods for detecting an illicit copy of a database. All of these involve writing dates into the database and keeping track of the file identifier in both the passwords and the database itself. Unfortunately, in many systems dates can be easily modified and the file identifier is preserved by certain copy mechanisms, such as backup and restore. A secure database would not be restorable from backup. In the event of a disk hardware failure or accidental deletion, the database would have to be recreated from scratch. Licenses would be unavailable for some time while new passwords were obtained from the vendors. We expect that a non-restorable database would be unacceptable to most users, and that an insecure database would be unacceptable to most vendors.

To provide availability we implemented static partitioning. The system administrator sets up each server to control a fraction of the licenses, and licenses may not be moved from one server to another. Servers do not communicate with one another; in the event of a network partition or hardware failure or network partition the licenses from intact databases are still available. Licenses are locked to one node, and only one server may run on a node at once, so there is no advantage to copying a database. The database can be backed up and restored like any other file. The NLS client library that is bound with the application is responsible for finding all of the servers and integrating the license information they provide.

3. Summary

NLS provides licensed concurrent access to applications in a node location independent manner. It has operated in a heterogeneous network of Digital Equipment Corp. VAX/VMS* machines and Sun and Apollo workstations. Our experience has been that the static partitioning scheme works quite well in practice. It provided sufficient availability, and the administration load is drastically reduced when compared to node locking each application on each node. We are continuing to look at other mechanisms to partition or replicate the database.

References

- [Bernstein 87] P. A. Bernstein, V. Hadzilacos, N. Goodman: "Concurrency control and recovery in database systems", Addison-Wesley Publishing Company, 1987.
- [Dineen 87] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, Geoffrey L. Wyant: "The network computing architecture and system: an environment for developing distributed applications", *Proceedings of the Usenix Association Summer Conference*, 1987.
- [NBS 77] National Bureau of Standards: "Data Encryption Standard", *Fed. Inf. Process. Stand. Publ. 46*: January 1977
- [Nelson 86] B. Nelson, A. Birrell: "Implementing Remote Procedure Calls", *ACM Trans. on Comp. Sys.* February 1986, pp 39-59.

*VAX and VMS are trademarks of Digital Equipment Corp.

Design of a General Purpose Memory Allocator for the 4.3BSD UNIX† Kernel

Marshall Kirk McKusick

Michael J. Karels

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The 4.3BSD UNIX kernel uses many memory allocation mechanisms, each designed for the particular needs of the utilizing subsystem. This paper describes a general purpose dynamic memory allocator that can be used by all of the kernel subsystems. The design of this allocator takes advantage of known memory usage patterns in the UNIX kernel and a hybrid strategy that is time-efficient for small allocations and space-efficient for large allocations. This allocator replaces the multiple memory allocation interfaces with a single easy-to-program interface, results in more efficient use of global memory by eliminating partitioned and specialized memory pools, and is quick enough that no performance loss is observed relative to the current implementations. The paper concludes with a discussion of our experience in using the new memory allocator, and directions for future work.

1. Kernel Memory Allocation in 4.3BSD

The 4.3BSD kernel has at least ten different memory allocators. Some of them handle large blocks, some of them handle small chained data structures, and others include information to describe I/O operations. Often the allocations are for small pieces of memory that are only needed for the duration of a single system call. In a user process such short-term memory would be allocated on the run-time stack. Because the kernel has a limited run-time stack, it is not feasible to allocate even moderate blocks of memory on it. Consequently, such memory must be allocated through a more dynamic mechanism. For example, when the system must translate a pathname, it must allocate a one kilobyte buffer to hold the name. Other blocks of memory must be more persistent than a single system call and really have to be allocated from dynamic memory. Examples include protocol control blocks that remain throughout the duration of the network connection.

Demands for dynamic memory allocation in the kernel have increased as more services have been added. Each time a new type of memory allocation has been required, a specialized memory allocation scheme has been written to handle it. Often the new memory allocation scheme has been built on top of an older allocator. For example, the block device subsystem provides a crude form of memory allocation through the allocation of empty buffers [Thompson78]. The allocation is slow because of the implied semantics of finding the oldest buffer, pushing its contents to disk if they are dirty, and moving physical memory into or out of the buffer to create the requested size. To reduce the overhead, a "new" memory allocator was built in 4.3BSD for name translation that allocates a pool of empty buffers. It keeps them on a free list so they can be quickly

†UNIX is a registered trademark of AT&T in the US and other countries.

allocated and freed [McKusick85].

This memory allocation method has several drawbacks. First, the new allocator can only handle a limited range of sizes. Second, it depletes the buffer pool, as it steals memory intended to buffer disk blocks to other purposes. Finally, it creates yet another interface of which the programmer must be aware.

A generalized memory allocator is needed to reduce the complexity of writing code inside the kernel. Rather than providing many semi-specialized ways of allocating memory, the kernel should provide a single general purpose allocator. With only a single interface, programmers do not need to figure out the most appropriate way to allocate memory. If a good general purpose allocator is available, it helps avoid the syndrome of creating yet another special purpose allocator.

To ease the task of understanding how to use it, the memory allocator should have an interface similar to the interface of the well-known memory allocator provided for applications programmers through the C library routines *malloc()* and *free()*. Like the C library interface, the allocation routine should take a parameter specifying the size of memory that is needed. The range of sizes for memory requests should not be constrained. The free routine should take a pointer to the storage being freed, and should not require additional information such as the size of the piece of memory being freed.

2. Criteria for a Kernel Memory Allocator

The design specification for a kernel memory allocator is similar to, but not identical to, the design criteria for a user level memory allocator. The first criterion for a memory allocator is that it make good use of the physical memory. Good use of memory is measured by the amount of memory needed to hold a set of allocations at any point in time. Percentage utilization is expressed as:

$$\text{utilization} = \frac{\text{requested}}{\text{required}}$$

Here, "requested" is the sum of the memory that has been requested and not yet freed. "Required" is the amount of memory that has been allocated for the pool from which the requests are filled. An allocator requires more memory than requested because of fragmentation and a need to have a ready supply of free memory for future requests. A perfect memory allocator would have a utilization of 100%. In practice, having a 50% utilization is considered good [Korn85].

Good memory utilization in the kernel is more important than in user processes. Because user processes run in virtual memory, unused parts of their address space can be paged out. Thus pages in the process address space that are part of the "required" pool that are not being "requested" need not tie up physical memory. Because the kernel is not paged, all pages in the "required" pool are held by the kernel and cannot be used for other purposes. To keep the kernel utilization percentage as high as possible, it is desirable to release unused memory in the "required" pool rather than to hold it as is typically done with user processes. Because the kernel can directly manipulate its own page maps, releasing unused memory is fast; a user process must do a system call to release memory.

The most important criterion for a memory allocator is that it be fast. Because memory allocation is done frequently, a slow memory allocator will degrade the system performance. Speed of allocation is more critical when executing in the kernel than in user code, because the kernel must allocate many data structure that user processes can allocate cheaply on their run-time stack. In addition, the kernel represents the platform on which all user processes run, and if it is slow, it will degrade the performance of every process that is running.

Another problem with a slow memory allocator is that programmers of frequently-used kernel interfaces will feel that they cannot afford to use it as their primary memory allocator. Instead they will build their own memory allocator on top of the original by maintaining their own pool of memory blocks. Multiple allocators reduce the efficiency with which memory is used. The kernel ends up with many different free lists of memory instead of a single free list from which all allocation can be drawn. For example, consider the case of two subsystems that need memory. If

they have their own free lists, the amount of memory tied up in the two lists will be the sum of the greatest amount of memory that each of the two subsystems has ever used. If they share a free list, the amount of memory tied up in the free list may be as low as the greatest amount of memory that either subsystem used. As the number of subsystems grows, the savings from having a single free list grow.

3. Existing User-level Implementations

There are many different algorithms and implementations of user-level memory allocators. A survey of those available on UNIX systems appeared in [Korn85]. Nearly all of the memory allocators tested made good use of memory, though most of them were too slow for use in the kernel. The fastest memory allocator in the survey by nearly a factor of two was the memory allocator provided on 4.2BSD originally written by Chris Kingsley at California Institute of Technology. Unfortunately, the 4.2BSD memory allocator also wasted twice as much memory as its nearest competitor in the survey.

The 4.2BSD user-level memory allocator works by maintaining a set of lists that are ordered by increasing powers of two. Each list contains a set of memory blocks of its corresponding size. To fulfill a memory request, the size of the request is rounded up to the next power of two. A piece of memory is then removed from the list corresponding to the specified power of two and returned to the requester. Thus, a request for a block of memory of size 53 returns a block from the 64-sized list. A typical memory allocation requires a roundup calculation followed by a linked list removal. Only if the list is empty is a real memory allocation done. The free operation is also fast; the block of memory is put back onto the list from which it came. The correct list is identified by a size indicator stored immediately preceding the memory block.

4. Considerations Unique to a Kernel Allocator

There are several special conditions that arise when writing a memory allocator for the kernel that do not apply to a user process memory allocator. First, the maximum memory allocation can be determined at the time that the machine is booted. This number is never more than the amount of physical memory on the machine, and is typically much less since a machine with all its memory dedicated to the operating system is uninteresting to use. Thus, the kernel can statically allocate a set of data structures to manage its dynamically allocated memory. These data structures never need to be expanded to accommodate memory requests; yet, if properly designed, they need not be large. For a user process, the maximum amount of memory that may be allocated is a function of the maximum size of its virtual memory. Although it could allocate static data structures to manage its entire virtual memory, even if they were efficiently encoded they would potentially be huge. The other alternative is to allocate data structures as they are needed. However, that adds extra complications such as new failure modes if it cannot allocate space for additional structures and additional mechanisms to link them all together.

Another special condition of the kernel memory allocator is that it can control its own address space. Unlike user processes that can only grow and shrink their heap at one end, the kernel can keep an arena of kernel addresses and allocate pieces from that arena which it then populates with physical memory. The effect is much the same as a user process that has parts of its address space paged out when they are not in use, except that the kernel can explicitly control the set of pages allocated to its address space. The result is that the "working set" of pages in use by the kernel exactly corresponds to the set of pages that it is really using.

A final special condition that applies to the kernel is that all of the different uses of dynamic memory are known in advance. Each one of these uses of dynamic memory can be assigned a type. For each type of dynamic memory that is allocated, the kernel can provide allocation limits. One reason given for having separate allocators is that no single allocator could starve the rest of the kernel of all its available memory and thus a single runaway client could not paralyze the system. By putting limits on each type of memory, the single general purpose memory allocator can provide the same protection against memory starvation.†

†One might seriously ask the question what good it is if "only" one subsystem within the kernel hangs if it is something like the network on a diskless workstation.

Memory statistics by bucket size			
Size	In Use	Free	Requests
128	329	39	3129219
256	0	0	0
512	4	0	16
1024	17	5	648771
2048	13	0	13
2049-4096	0	0	157
4097-8192	2	0	103
8193-16384	0	0	0
16385-32768	1	0	1

Memory statistics by type				
Type	In Use	Mem Use	High Use	Requests
mbuf	6	1K	17K	3099066
devbuf	13	53K	53K	13
socket	37	5K	6K	1275
pcb	55	7K	8K	1512
routetbl	229	29K	29K	2424
fragtbl	0	0K	1K	404
zombie	3	1K	1K	24538
namei	0	0K	5K	648754
ioctlops	0	0K	1K	12
superblk	24	34K	34K	24
temp	0	0K	8K	258

Figure 1. One day memory usage on a Berkeley time-sharing machine

Figure 1 shows the memory usage of the kernel over a one day period on a general timesharing machine at Berkeley. The "In Use", "Free", and "Mem Use" fields are instantaneous values; the "Requests" field is the number of allocations since system startup; the "High Use" field is the maximum value of the "Mem Use" field since system startup. The figure demonstrates that most allocations are for small objects. Large allocations occur infrequently, and are typically for long-lived objects such as buffers to hold the superblock for a mounted file system. Thus, a memory allocator only needs to be fast for small pieces of memory.

5. Implementation of the Kernel Memory Allocator

In reviewing the available memory allocators, none of their strategies could be used without some modification. The kernel memory allocator that we ended up with is a hybrid of the fast memory allocator found in the 4.2BSD C library and a slower but more-memory-efficient first-fit allocator.

Small allocations are done using the 4.2BSD power-of-two list strategy; the typical allocation requires only a computation of the list to use and the removal of an element if it is available, so it is quite fast. Macros are provided to avoid the cost of a subroutine call. Only if the request cannot be fulfilled from a list is a call made to the allocator itself. To ensure that the allocator is always called for large requests, the lists corresponding to large allocations are always empty. Appendix A shows the data structures and implementation of the macros.

Similarly, freeing a block of memory can be done with a macro. The macro computes the list on which to place the request and puts it there. The free routine is called only if the block of memory is considered to be a large allocation. Including the cost of blocking out interrupts,

the allocation and freeing macros generate respectively only nine and sixteen (simple) VAX instructions.

Because of the inefficiency of power-of-two allocation strategies for large allocations, a different strategy is used for allocations larger than two kilobytes. The selection of two kilobytes is derived from our statistics on the utilization of memory within the kernel, that showed that 95 to 98% of allocations are of size one kilobyte or less. A frequent caller of the memory allocator (the name translation function) always requests a one kilobyte block. Additionally the allocation method for large blocks is based on allocating pieces of memory in multiples of pages. Consequently the actual allocation size for requests of size $2 \times \text{pagesize}$ or less are identical.[†] In 4.3BSD on the VAX, the (software) page size is one kilobyte, so two kilobytes is the smallest logical cutoff.

Large allocations are first rounded up to be a multiple of the page size. The allocator then uses a first-fit algorithm to find space in the kernel address arena set aside for dynamic allocations. Thus a request for a five kilobyte piece of memory will use exactly five pages of memory rather than eight kilobytes as with the power-of-two allocation strategy. When a large piece of memory is freed, the memory pages are returned to the free memory pool, and the address space is returned to the kernel address arena where it is coalesced with adjacent free pieces.

Another technique to improve both the efficiency of memory utilization and the speed of allocation is to cluster same-sized small allocations on a page. When a list for a power-of-two allocation is empty, a new page is allocated and divided into pieces of the needed size. This strategy speeds future allocations as several pieces of memory become available as a result of the call into the allocator.

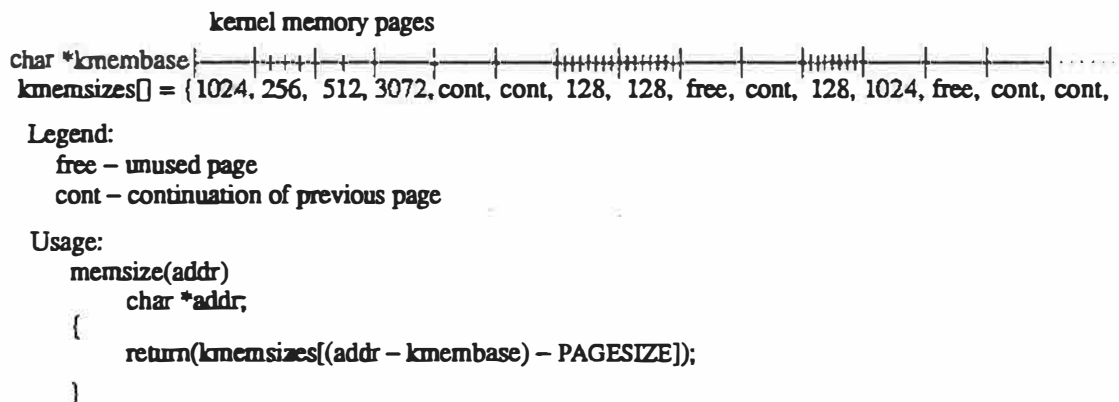


Figure 2. Calculation of allocation size

Because the size is not specified when a block of memory is freed, the allocator must keep track of the sizes of the pieces it has handed out. The 4.2BSD user-level allocator stores the size of each block in a header just before the allocation. However, this strategy doubles the memory requirement for allocations that require a power-of-two-sized block. Therefore, instead of storing the size of each piece of memory with the piece itself, the size information is associated with the memory page. Figure 2 shows how the kernel determines the size of a piece of memory that is being freed, by calculating the page in which it resides, and looking up the size associated with that page. Eliminating the cost of the overhead per piece improved utilization far more than

[†]To understand why this number is $2 \times \text{pagesize}$ one observes that the power-of-two algorithm yields sizes of 1, 2, 4, 8, ... pages while the large block algorithm that allocates in multiples of pages yields sizes of 1, 2, 3, 4, ... pages. Thus for allocations of sizes between one and two pages both algorithms use two pages; it is not until allocations of sizes between two and three pages that a difference emerges where the power-of-two algorithm will use four pages while the large block algorithm will use three pages.

expected. The reason is that many allocations in the kernel are for blocks of memory whose size is exactly a power of two. These requests would be nearly doubled if the user-level strategy were used. Now they can be accommodated with no wasted memory.

The allocator can be called both from the top half of the kernel, which is willing to wait for memory to become available, and from the interrupt routines in the bottom half of the kernel that cannot wait for memory to become available. Clients indicate their willingness (and ability) to wait with a flag to the allocation routine. For clients that are willing to wait, the allocator guarantees that their request will succeed. Thus, these clients can need not check the return value from the allocator. If memory is unavailable and the client cannot wait, the allocator returns a null pointer. These clients must be prepared to cope with this (hopefully infrequent) condition (usually by giving up and hoping to do better later).

6. Results of the Implementation

The new memory allocator was written about a year ago. Conversion from the old memory allocators to the new allocator has been going on ever since. Many of the special purpose allocators have been eliminated. This list includes `calloc()`, `wmemall()`, and `zmemall()`. Many of the special purpose memory allocators built on top of other allocators have also been eliminated. For example, the allocator that was built on top of the buffer pool allocator `geteblk()` to allocate pathname buffers in `namei()` has been eliminated. Because the typical allocation is so fast, we have found that none of the special purpose pools are needed. Indeed, the allocation is about the same as the previous cost of allocating buffers from the network pool (`mbufs`). Consequently applications that used to allocate network buffers for their own uses have been switched over to using the general purpose allocator without increasing their running time.

Quantifying the performance of the allocator is difficult because it is hard to measure the amount of time spent allocating and freeing memory in the kernel. The usual approach is to compile a kernel for profiling and then compare the running time of the routines that implemented the old abstraction versus those that implement the new one. The old routines are difficult to quantify because individual routines were used for more than one purpose. For example, the `geteblk()` routine was used both to allocate one kilobyte memory blocks and for its intended purpose of providing buffers to the filesystem. Differentiating these uses is often difficult. To get a measure of the cost of memory allocation before putting in our new allocator, we summed up the running time of all the routines whose exclusive task was memory allocation. To this total we added the fraction of the running time of the multi-purpose routines that could clearly be identified as memory allocation usage. This number showed that approximately three percent of the time spent in the kernel could be accounted to memory allocation.

The new allocator is difficult to measure because the usual case of the memory allocator is implemented as a macro. Thus, its running time is a small fraction of the running time of the numerous routines in the kernel that use it. To get a bound on the cost, we changed the macro always to call the memory allocation routine. Running in this mode, the memory allocator accounted for six percent of the time spent in the kernel. Factoring out the cost of the statistics collection and the subroutine call overhead for the cases that could normally be handled by the macro, we estimate that the allocator would account for at most four percent of time in the kernel. These measurements show that the new allocator does not introduce significant new run-time costs.

The other major success has been in keeping the size information on a per-page basis. This technique allows the most frequently requested sizes to be allocated without waste. It also reduces the amount of bookkeeping information associated with the allocator to four kilobytes of information per megabyte of memory under management (with a one kilobyte page size).

7. Future Work

Our next project is to convert many of the static kernel tables to be dynamically allocated. Static tables include the process table, the file table, and the mount table. Making these tables dynamic will have two benefits. First, it will reduce the amount of memory that must be statically allocated at boot time. Second, it will eliminate the arbitrary upper limit imposed by the current

static sizing (although a limit will be retained to constrain runaway clients). Other researchers have already shown the memory savings achieved by this conversion [Rodriguez88].

Under the current implementation, memory is never moved from one size list to another. With the 4.2BSD memory allocator this causes problems, particularly for large allocations where a process may use a quarter megabyte piece of memory once, which is then never available for any other size request. In our hybrid scheme, memory can be shuffled between large requests so that large blocks of memory are never stranded as they are with the 4.2BSD allocator. However, pages allocated to small requests are allocated once to a particular size and never changed thereafter. If a burst of requests came in for a particular size, that size would acquire a large amount of memory that would then not be available for other future requests.

In practice, we do not find that the free lists become too large. However, we have been investigating ways to handle such problems if they occur in the future. Our current investigations involve a routine that can run as part of the idle loop that would sort the elements on each of the free lists into order of increasing address. Since any given page has only one size of elements allocated from it, the effect of the sorting would be to sort the list into distinct pages. When all the pieces of a page became free, the page itself could be released back to the free pool so that it could be allocated to another purpose. Although there is no guarantee that all the pieces of a page would ever be freed, most allocations are short-lived, lasting only for the duration of an open file descriptor, an open network connection, or a system call. As new allocations would be made from the page sorted to the front of the list, return of elements from pages at the back would eventually allow pages later in the list to be freed.

Two of the traditional UNIX memory allocators remain in the current system. The terminal subsystem uses *clists* (character lists). That part of the system is expected to undergo major revision within the the next year or so, and it will probably be changed to use *mbufs* as it is merged into the network system. The other major allocator that remains is *getblk()*, the routine that manages the filesystem buffer pool memory and associated control information. Only the filesystem uses *getblk()* in the current system; it manages the constant-sized buffer pool. We plan to merge the filesystem buffer cache into the virtual memory system's page cache in the future. This change will allow the size of the buffer pool to be changed according to memory load, but will require a policy for balancing memory needs with filesystem cache performance.

8. Acknowledgments

In the spirit of community support, we have made various versions of our allocator available to our test sites. They have been busily burning it in and giving us feedback on their experiences. We acknowledge their invaluable input. The feedback from the Usenix program committee on the initial draft of our paper suggested numerous important improvements.

9. References

- Korn85 David Korn, Kiem-Phong Vo, "In Search of a Better Malloc" *Proceedings of the Portland Usenix Conference*, pp 489-506, June 1985.
- McKusick85 M. McKusick, M. Karels, S. Leffler, "Performance Improvements and Functional Enhancements in 4.3BSD" *Proceedings of the Portland Usenix Conference*, pp 519-531, June 1985.
- Rodriguez88 Robert Rodriguez, Matt Koehler, Larry Palmer, Ricky Palmer, "A Dynamic UNIX Operating System" *Proceedings of the San Francisco Usenix Conference*, June 1988.
- Thompson78 Ken Thompson, "UNIX Implementation" *Bell System Technical Journal*, volume 57, number 6, pp 1931-1946, 1978.

10. Appendix A - Implementation Details

```

/*
 * Constants for setting the parameters of the kernel memory allocator.
 *
 * 2 ** MINBUCKET is the smallest unit of memory that will be
 * allocated. It must be at least large enough to hold a pointer.
 *
 * Units of memory less or equal to MAXALLOCSAVE will permanently
 * allocate physical memory; requests for these size pieces of memory
 * are quite fast. Allocations greater than MAXALLOCSAVE must
 * always allocate and free physical memory; requests for these size
 * allocations should be done infrequently as they will be slow.
 * Constraints: CLBYTES <= MAXALLOCSAVE <= 2 ** (MINBUCKET + 14)
 * and MAXALLOCSIZE must be a power of two.
 */
#define MINBUCKET          4                /* 4 => min allocation of 16 bytes */
#define MAXALLOCSAVE      (2 * CLBYTES)

/*
 * Maximum amount of kernel dynamic memory.
 * Constraints: must be a multiple of the pagesize.
 */
#define MAXKMEM             (1024 * PAGE_SIZE)

/*
 * Arena for all kernel dynamic memory allocation.
 * This arena is known to start on a page boundary.
 */
extern char kmembase[MAXKMEM];

/*
 * Array of descriptors that describe the contents of each page
 */
struct kmemsizes {
    short    ks_idx;           /* bucket index, size of small allocations */
    u_short  ks_pagecnt;      /* for large allocations, pages allocated */
} kmemsizes[MAXKMEM / PAGE_SIZE];

/*
 * Set of buckets for each size of memory block that is retained
 */
struct kmembuckets {
    caddr_t kb_next;          /* list of free blocks */
} bucket[MINBUCKET + 16];

```

```

/*
 * Macro to convert a size to a bucket index. If the size is constant,
 * this macro reduces to a compile time constant.
 */
#define MINALLOCSIZE      (1 << MINBUCKET)
#define BUCKETINDX(size) \
    (size) <= (MINALLOCSIZE * 128) \
    ? (size) <= (MINALLOCSIZE * 8) \
    ? (size) <= (MINALLOCSIZE * 2) \
    ? (size) <= (MINALLOCSIZE * 1) \
    ? (MINBUCKET + 0) \
    : (MINBUCKET + 1) \
    : (size) <= (MINALLOCSIZE * 4) \
    ? (MINBUCKET + 2) \
    : (MINBUCKET + 3) \
    : (size) <= (MINALLOCSIZE * 32) \
    ? (size) <= (MINALLOCSIZE * 16) \
    ? (MINBUCKET + 4) \
    : (MINBUCKET + 5) \
    : (size) <= (MINALLOCSIZE * 64) \
    ? (MINBUCKET + 6) \
    : (MINBUCKET + 7) \
    : (size) <= (MINALLOCSIZE * 2048) \
    /* etc ... */

/*
 * Macro versions for the usual cases of malloc/free
 */
#define MALLOC(space, cast, size, flags) { \
    register struct knembuckets *kbp = &bucket[BUCKETINDX(size)]; \
    long s = splimp(); \
    if (kbp->kb_next == NULL) { \
        (space) = (cast)malloc(size, flags); \
    } else { \
        (space) = (cast)kbp->kb_next; \
        kbp->kb_next = *(caddr_t *) (space); \
    } \
    splx(s); \
}

#define FREE(addr) { \
    register struct knembuckets *kbp; \
    register struct knemsize *ksp = \
        &knemsize[(((addr) - knembase) / PAGE_SIZE)]; \
    long s = splimp(); \
    if (1 << ksp->ks_indx > MAXALLOCSAVE) { \
        free(addr); \
    } else { \
        kbp = &bucket[ksp->ks_indx]; \
        *(caddr_t *) (addr) = kbp->kb_next; \
        kbp->kb_next = (caddr_t) (addr); \
    } \
    splx(s); \
}

```


A Dynamic UNIX¹ Operating System

Robert Rodriguez

Matt Koehler

Larry Palmer

Ricky Palmer

{rr, mjk, lp, rsp}@decvax.dec.com

Digital Equipment Corporation

ULTRIX² Engineering Group

110 Spitbrook Road

Nashua, NH 03062

ABSTRACT

Most of the data structures in a traditional UNIX kernel are implemented as static tables. A configuration file or data available at boot time determines the size of these tables. This paper discusses the changes made to allow such tables to become linked lists of structures. These linked lists expand or contract as system activity changes.

This work is based on the ULTRIX V2.0 kernel. As a point of reference, most of the data structures and algorithms in the ULTRIX V2.0 kernel parallel the Berkeley 4.3 kernel (as opposed to the SYSTEM V kernel). Most modifications made in the research discussed in this paper can be directly transported to the 4.3BSD kernel. Many changes can be implemented in a System V kernel.

1. THE PROBLEM OF TABLES

The original design of the UNIX Operating System was targeted for a miniscule machine by today's standards [Ritch1984a, Ritch1978a, Thomp1978a]. The first port to a PDP-11 contained only 16 kilobytes (KB) of memory for the kernel and 8 KB of memory for users. Its disks held 512 KB and files sizes were limited to 64 KB. The small memory of the PDP-11 required extremely tight control of resources with limited table sizes for each structure. It is a tribute to the design of the UNIX Operating System that few of its internal structures and algorithms have changed.

Today, machines contain 2 MB to 512 MB of memory, may be diskless, or have terabytes of disk storage. These machines connect to high speed networks and devices. Since configurations of machines vary widely, the UNIX kernel needs a more dynamic and flexible approach to managing internal data structures.

The prototype kernel removes restrictions present in a statically allocated table-based system. The kernel allocates most data structures as needed (rather than at boot time). This paper details the work involved to move to a dynamic kernel. Algorithmic changes, structure changes, performance and future work are also discussed.

2. A FAST MEMORY ALLOCATOR FOR THE KERNEL

The prototype requires an efficient and fast allocator of free kernel memory. Fortunately, the 4.3BSD release contains such an allocator [McKusi1988a]. The allocator uses a simple power-of-two bucket scheme where each bucket contains a linked list of identically sized pieces of free memory. Buckets in the prototype system hold memory in sizes ranging from 16 bytes to 256 KB.³ The interface to retrieve a piece of free memory uses a macro

¹ UNIX is a registered trademark and System V is a trademark of A.T. & T. Information Systems.

² ULTRIX, MicroVAX, VAX, PDP-11, and DEQNA are trademarks of Digital Equipment Corporation.

³ Both minimum and maximum allocation is tunable.

expanding to only 9 VAX instructions. This assumes a piece of free memory exists on the bucket (which is expected after the system is up or if the buckets are pre-populated).

2.1. A Few Additions to the Allocator

The allocator from 4.3BSD needs additional functionality. Extra protection is added around allocations one cluster (1 KB) or larger to catch illegal accesses. This helps catch bugs in the prototype system. Some device drivers require physically contiguous memory; the allocator optionally returns contiguous memory pages. Because the network code passes large pieces of unmodified memory (larger than 1 KB), an option called MEMDUP has been added. Instead of writing specialized code to handle memory passing, the memory allocator keeps a reference count. Calls to the memory free routine decrement the count. Removing the final reference places the free memory in the appropriate bucket.

Heavy use of the allocator can place a lot of memory in the buckets. The prototype system contains a memory scrubber that trims each bucket back to user-settable limits. When possible, free memory smaller than a cluster is coalesced into clusters. The free clusters return to the coremap making the memory available to user processes.

3. STEPPING INTO THE DYNAMIC WORLD

The rest of this paper describes the changes to allow the dynamic allocation of process, file, mount, text, and gnode (called inode in 4.3BSD) tables. Further, the paper presents modifications to make the buffer cache, the network code, and device drivers dynamic.

3.1. The Process Table

System responsiveness is sensitive to the performance of the *fork* and *exec* system calls. Therefore, much work is done to make process creation fast and care is taken in managing process elements (the process slot, associated linked lists, coremap entries, and the text slot).

Further, dependencies exist throughout the kernel on the address of `proc[0]` (the swapper), `proc[1]` (the init process), and `proc[2]` (the pagedaemon). References to table indices exist throughout the UNIX kernel. For example, in the static system, the coremap (`cmap`) structure holds an index (`c_idx`) into the process and text tables. The knowledge that elements are kept in tables affects coding techniques. Therefore, a brief discussion of past implementation follows.

Scattered throughout the old table-based kernel is the assumption that if `p1` is a pointer to a `struct proc` and `proc` is a pointer to the head of the table, then `i = p1 - proc` can be used as a table index. This index `i`, is usually a small positive integer and can be stored away in another data structure. Saving the table index rather than a pointer to the element may save a few bytes. Later, when the code wishes to retrieve the process pointer, it can simply refer to `&proc[i]`. Almost every table-base structure in the kernel depends on this construct. Further, code like `for(p = proc; p < &proc[NPROC]; p++)` do not work for non-contiguous elements.

In making the process table dynamically allocated, the pidhash list is eliminated. The hash list relies on the table index. At present, the prototype has no process pidhash list although designing a new algorithm or calculating pseudo-indices would not be difficult. The `cmap` structure replaces `c_idx` with a pointer. This increases the `cmap` by 4 bytes.⁴ Even though a pointer is added to the coremap, memory use is more efficient since static tables are replaced by dynamic allocation of individual elements. The process structure also needs some special handling of processes 0, 1, and 2. In the system initialization code, it is easier declaring three process structures and pointers as 'bss', and linking the structures in to the

⁴ Adding 4 bytes to the size of the coremap seems irrelevant. However, on a 7 MB MicroVAX, there are approximately 5300 elements in the coremap. This causes the table to grow by 21 KB.

active process list.

A few other points are worth mentioning. The variable `nproc` is the number of process allocated (in the static system, it is the number of elements in the process table). Several other structures in the process structure related to System V shared memory are removed from the process structure and allocated as needed. Only two lists exist for procs, `allproc` and `zombproc`. `Allproc` is the list of runnable processes and `zombproc` is the list of processes exiting and waiting for their parent to do the `wait` system call. Since the process structure has lists of active and zombie processes, no new fields are added. There are some modifications to create a new process and destroy an exiting one. Six macros help kernel programming to handle the linked lists. These macros are shown in Table 1. In general, macros similar to these exist for all the dynamically allocated structures.

Macro	Explanation
<code>P_ADDALL(p)</code>	Add process <code>p</code> to <code>allproc</code> list
<code>P_REMALL(p)</code>	Remove process <code>p</code> from <code>allproc</code> list
<code>P_ADDZOM(p)</code>	Add process <code>p</code> to <code>zombproc</code> list
<code>P_REMZOM(p)</code>	Remove process <code>p</code> from <code>zombproc</code> list
<code>FORALLPROC(p)</code>	For loop, <code>p</code> = all processes on <code>allproc</code>
<code>FORALLZOMB(p)</code>	For loop, <code>p</code> = all processes on <code>zombproc</code>

TABLE 1 – Process Manipulation Macros

3.2. The File Table

In the static system, access to the file table is similar to access to the process table. The list is often searched with code similar to:

```
for(fp = file; fp < &file[NFILE]; fp++)
```

Figures 2 and 3 show the old and new file structures.

```
struct    file {
    int    f_flag;
    short  f_type;           /* descriptor type */
    short  f_count;         /* reference count */
    short  f_msgcount;      /* message queue refs */
    struct fileops {
        /* return function arguments */
        int    (*fo_rw)(      /* fp,uio,rw */ );
        int    (*fo_ioctl)(  /* fp,com,data,cred*/ );
        int    (*fo_select)( /* fp,which */ );
        int    (*fo_close)(  /* fp */ );
    }    *f_ops;
    caddr_t f_data;         /* gnode */
    off_t f_offset;         /* r/w location */
    struct ucred *f_cred;   /* user permissions */
};
```

FIGURE 2 – Old File Structure

The file structure now contains a pair of pointers maintaining a forward and backward list. Further, many cases have been found where file pointers are not consistently used. By writing the macros `F_ADDQ(fp)` and `F_REMQ(fp)`, and searching for either "decrement the count on the file pointer" or "free the credentials" or "set the count of the file pointer to

```

struct filelist {
    struct file      *fl_next;          /* next on list */
    struct file      *fl_prev;          /* previous on list */
} filelist;

struct file {
    struct filelist f_list;              /* linked list */
    /* The remainder of the old structure */
};

```

FIGURE 3 – New File Structure

zero", several dangling references are corrected.

3.3. The Text Table

The text entries are allocated from the routine *xalloc* and returned by *xfree*. In the static system, the text table is created at boot time and all entries are placed on the free list of texts. Allocation is done by taking a text table entry off the free list of texts and attaching it to the requisite gnode. This gnode has its count artificially incremented to avoid reuse. At free time, the text is returned to the end of the free list. Even though the text entry is attached to the free list, the inflated count on the gnode remains. It should be noted that a gnode and a text slot are attached to each other and gnodes are cached. Therefore, the static system creates a cache of text entries. The binding between text and gnode is only broken when the text slot is used for a different text.

The text cache is desirable. When reclaiming a text, allocating swap space and setting up the space for the executable image is not necessary. In allocating entries, a tradeoff must be made between returning the memory associated with the text slot at *xfree* time (and therefore discarding the cache), and never returning the text entry. When the maximum number of dynamically allocated texts approaches the maximum number of gnodes, executable images can consume all available gnodes. To circumvent this problem, when the count on the text slot is decremented to zero, the gnode count is decremented. This allows the gnode either to be reclaimed from the cache (which then allows the text entry to be reclaimed), or to have the gnode used for another file. Re-using the gnode causes the text slot to be returned to the malloc pool.

3.4. The Mount Table

In the static system, the mount table is allocated at boot time. The constant *NMOUNT* restricts the number of elements in the mount table. The number of bits allocated to the *c_mdev* field in the *cmap* structure further limits the size of this table. The static system allows 8 bits for the *c_mdev* field. This permits the static system to mount 254 file systems (there are two reserved numbers: *MSWAPX* indicates the *cmap* entry points to the swap device and *NODEV* indicates an unused entry). In the prototype system, *c_mdev* is a single bit indicating a page exists on the swap surface. The device number can be fetched from the attached gnode. The *cmap* has been effectively purged of its need for mounted file system indices.

When mounting a file system, the code in the static system searches linearly through the mount table to find an unused entry. In the prototype system, mount structures are allocated as needed. The prototype system then calculates a unique index used by the file system quota code. Finally, the mount is attempted. If this action succeeds, the new mount structure is linked into the list of active mounts. Stepping through the list of mounted file systems is simplified since all structures in the list refer to active mounts. When unmounting a file system, the operating system unlinks the mount structure from the active mounts and returns the space to the system.

The static system has two references to the old mount table. The first is the `cmap` structure which is discussed in the beginning of this section. The second reference is in the file system quota code. The quota code has implicit knowledge of how quota tables are matched to mount table elements (the index previously was calculated as `mp - mount`). The mount structure's unique index preserves the quota functionality without rewriting it. This allows the binding between the quota arrays and the dynamic mount structures to continue. Time and effort will free the quota system of this index problem.

3.5. The Gnode Table

In the static system, the gnode structures are kept in a table. The number of gnodes is then defined at boot time (by calculations using `MAXUSERS` and `nproc`). This table holds both active and inactive references to files. The gnode data is cached so inactive references can be quickly reclaimed.

The problem then is how to maintain the cache of inactive gnodes without causing all of memory to be consumed with gnodes. Various strategies have been devised including bounding the number of gnodes by some percentage of physical memory and allowing a configurable parameter. Since it is desirable to manage the size of the gnode cache, the latter strategy has been chosen.

The allocation policy is simple. If a request for a particular on-disk (or on-network) device and inode number cannot be fetched from cache, an attempt is made to create a new gnode. Gnode creation occurs if the number of gnodes does not exceed the configurable limit and if there is memory available for the creation. Destroying gnodes is done through a system call. Destruction includes finding inactive gnodes, removing any dangling references to the gnode (such as an attached text) and returning the memory to the system.⁵

3.6. The Buffer Cache

The buffer cache is memory the operating system uses to hold file system blocks. As with any cache, its purpose is to reduce the frequency of sending transactions to some underlying mechanism. In the case of the buffer cache, the purpose is to reduce I/O requests to disk drives. The cache is broken into two pieces: buffer space holding the actual data, and buffer headers describing that data. The buffer headers are further broken down. Headers may appear on four different lists: locked, least recently used (`lru`), age,⁶ and empty. The locked list contains buffers that must be kept permanently. The `lru` list contains the actual cache. The age list contains buffers marked for delayed writing (among other things). The empty list contains buffers with no attached buffer space.

The goal with dynamically allocating the buffer cache is to have it consume most of free memory (i.e., memory potentially available to user processes). However, interactions with the virtual memory subsystem can cause significant impact on performance. Therefore, the buffer cache must release memory when the system anticipates (or requires) more free memory.

3.6.1. Buffer Allocation

This section describes how the old system and the new system create and maintain the buffer headers and the buffer space.

3.6.1.1. Allocation in the Static System

There are two aspects to allocating the buffer cache: allocating buffer headers and allocating buffer space. In previous versions of the operating system, buffer headers and space are allocated at boot time. Typically, 10% of physical memory is allocated for buffer space

⁵ The system call has been designed and but not yet implemented.

⁶ The locked list is never used; further the age and `lru` lists are really the head and tail of the same list.

and then one buffer header is allocated for every 2 KB of buffer space.

A call to *getnewbuf* returns a buffer header. This routine walks backward from the age list to the lru list looking for a buffer header. If the header is marked for delayed write, the I/O is started and *getnewbuf* searches for a new buffer header. Buffer space is requested by calling *allocbuf*. If the buffer size is shrinking, the routine attempts to place the excess space on the first buffer in the empty list. If more buffer space is to be attached to the header, the routine asks for a new buffer header (from *getnewbuf*) and takes the buffer space attached to the header. This action is continued until enough memory can be obtained to fulfill the request. Any buffer headers having all their space taken are inserted on the empty list.

3.6.1.2. Allocation in the Dynamic System

In the dynamic kernel, only the buffer hash lists are created at boot time. Buffer headers and buffer space are allocated and destroyed while the system is running. The number of buffer headers is bounded on the bottom by $\text{physmem} / (10 * \text{MAXBSIZE})^7$ and on the top by the configuration parameter *max_bhead_alloc*. Buffer space is bounded by $\text{physmem} / 10$ and the configuration parameter *max_bsize_alloc*.

Buffer headers are still obtained from *getnewbuf*. This routine checks to see if the number of buffer headers is less than *max_bhead_alloc*. If not, a new header is created, placed on the age list, inserted in the buffer list, and returned to the caller. If a buffer header cannot be created, the routine reverts to its old behavior of walking through the age and lru lists looking for buffer headers.

Likewise, buffer space is still obtained from *allocbuf*. *Allocbuf* now allocates space only in multiples of *MAXBSIZE*. If called with a buffer header having no space, the routine checks to see if the buffer cache is smaller than *max_bsize_alloc* and *freemem* is larger than *lotsfree*. If *freemem* is larger than *lotsfree*, the system does not need to page processes out to obtain memory and can presume that there is no outstanding need for physical memory. If no space is obtainable, the routine walks backward from the age list looking for a buffer to fulfill its request. If the buffer is dirty, it is written and the search for a buffer header continues. When an appropriate buffer header is found, the space is attached to the old buffer. The buffer header having its space reclaimed is removed from all lists and destroyed. Unlike the static system, *allocbuf* cannot call *getnewbuf* to obtain a buffer header. The call to *getnewbuf* can result in the creation of a new buffer header. Since the reason for requesting a buffer header is to reclaim space from it, creation of a new buffer header is not desired.

3.6.2. Buffer Deallocation

The old system had no method for deallocating buffer headers or buffer space. Empty buffer headers were placed on the empty list. Unused buffer space was attached to a buffer and the buffer was marked as invalid. The new system has no empty buffer list. When a buffer space is no longer needed, the space is returned to the allocator and the header is unlinked from all lists and freed.

3.6.3. Buffer Cache Algorithmic Changes

As previously discussed, the empty buffer list no longer exists. Another linked list is added to buffer headers. It allows the examination of the entire buffer cache. In the static kernel, each header could be examined by walking through the table element by element. This is no longer possible since the buffer headers are not kept in an array. It is also not sufficient to walk through the locked, lru, and age lists. Buffer headers having pending requests to devices do not appear in any of these lists. Finally, *allocbuf* only allocates space in multiples of *MAXBSIZE*. This is done since taking pages (by *pagemove*) from the kernel memory

⁷ *MAXBSIZE* is 8 KB.

allocator is forbidden.

There is a noticeable effect from the *sync* system call.⁸ When the number of buffer headers becomes large (several thousand on a 128 MB machine), *sync* causes the system to "freeze" for a couple of seconds while the device drivers are inundated with write requests. Changing asynchronous writes to synchronous writes avoids the problem (with a loss of performance). This modification causes the *write* system call to lose approximately 27% of its throughput. Other solutions are possible (including not syncing disks during normal operation or running several update daemons).

3.6.4. Buffer Cache Interaction with Virtual Memory

A tradeoff exists between consuming physical memory for the buffer cache and having it available for user processes. If `max_bsize_alloc` approaches physical memory, the buffer cache tends to dominate memory. This causes a drastic amount of paging activity. This is unacceptable.

During the early phases of the prototype, when the system transitioned to multi-user, the buffer cache consumed over 7 MB of memory in a 9 MB system. The goal of consuming most of free memory for the buffer cache had been achieved. Unfortunately, to get the password prompt for *login*, the system had to page in *X*,⁹ the terminal emulator, and *login*. This is an undesirable side effect of a relatively unrestricted, dynamically allocated, buffer cache.

To avoid the problem, the virtual memory subsystem calls *bufscleanup* to circumvent unnecessary paging or swapping activity. This routine walks from the age list to the lru list returning buffer space and buffer headers. It examines each header checking to make sure the header is not busy, locked, involved in physical I/O, or marked as delayed write. Buffer headers meeting these requirements are then destroyed and the attached space is returned to the system. This continues until enough memory is returned. The decision to not write delayed write buffer headers attempts to alleviate disk activity. Since the cleanup routine tends to return buffers in a least recently used fashion, the impact on the system is not noticeable. Several algorithms could be used to decide when to reclaim memory from the buffer cache. A system call could select these alternatives.

3.7. The Network

This section describes the changes necessary to make the network subsystem dynamic. An overview of the current Berkeley networking implementation is found in [Leffl1983a]. A discussion of changes needed to use the new allocator is presented. This will be followed by the lessons learned and what needs to be completed in the network.

3.7.1. 4.3BSD Networking Overview

Network memory allocation has not drastically changed since the introduction of networking code in 4.2BSD. The system uses small parts of a reserved pool (small mbufs) to hold data in the system. Mbufs pass from protocol layer to protocol layer. Each small mbuf contains a header region (to keep track of the mbuf while in use) and a local data region (to hold whatever data needs to be stored). Since small mbufs provide a limited amount of storage, large pieces of memory, large mbufs, can be coupled with small mbufs to provide for larger virtually contiguous storage. Both types of mbufs begin as a map of page table entries to which real memory is attached and segmented as the need arises. Both small and large mbufs are linked to a free list when not in use. Small mbufs are `MLEN` in size (currently 128 bytes) and large mbufs are `CLBYTES` (currently 1024 bytes). Finally, a reference count is maintained on large mbufs (in an array created at load time in the static system) so it may be

⁸ The problem with *sync* exists in the static system also. Dynamically allocating the buffer cache aggravates the issue because potentially many more buffers exist in the system.

⁹ *X* is a trademark of Massachusetts Institute of Technology.

coupled with many small mbufs as a quick form of copy.

3.7.2. Problems with the Current Network Allocation Policy

There are problems arising from the way mbufs are used in the system. The most problematic is the static upper limit on memory in the resource pool. When the resource pool is exhausted, the system panics. Thus, a temporary surge in network activity can have disastrous results. Another problem is large and small mbufs are used to maintain system structures which may not be part of the network.

Since an efficient kernel memory allocator has not been available, many areas in the kernel use mbufs as temporary storage. The static system uses mbufs to hold information on sockets, protocol control blocks, network interface information, and routing information. This leads to a dependency between the size of mbufs and how they are used (e.g., a kernel developer must know a socket structure will fit in `MSIZE` bytes of a small mbuf).

The first goal is to allow the network to be more dynamic in its use of memory. The second goal is to remove the need for mbufs in other parts of the kernel.

3.7.3. Network Interaction with New Allocator

Since the new allocator is fast, the new allocator allocates and frees all network resources (i.e., free lists of small and large mbufs are not maintained). Other network structures like sockets and protocol control blocks require only minor changes to allow usage of the new allocator. The main changes are calls to the allocator instead of the small mbuf "get" routine and calls to the allocator free routine when the memory is no longer needed. Usage of the new allocator for routing structures requires a little more work as some utilities "know" routing information exists in a linked list of small mbufs.

As mentioned earlier, the new allocator keeps a reference count on large allocations (an allocation larger than `CLBYTES`). This reference count allows any large allocation to be virtually copied. When the reference count becomes zero as a result of subsequent frees, the memory is returned to the free pool.

Finally, the way small mbufs point to large mbufs has to change. The old allocation method assumes large mbufs are always located at some positive offset in memory from the address of the small mbuf. To locate a large mbuf's data area, a small positive offset is added to the address of the small mbuf. The new allocator invalidates this assumption so the offset becomes a pointer.

3.7.4. Network Performance

Changing to the new allocator has no measurable change on the speed of the network. The use of free lists by the old allocation method is duplicated by the bucket lists in the new allocator. Allocating and freeing resources using the new allocator is at least as efficient as using the mbuf allocation policy.

The old allocation method wastes memory. It populates the free lists as the system is booted. This wastes memory if the network is either never used or has only minimal traffic. The old allocation method holds as much memory as needed to handle the greatest peak in network activity. This might allow a single peak in network traffic to consume a vast amount of memory, which may never again be needed. Finally, fitting dynamic system structures into small mbufs wastes memory. For instance, a protocol control block consumes in 36 bytes in a 128-byte small mbuf.

The new allocator helps fix these problems. The system boots with no memory allocated to the network. As peaks in traffic occur, the allocator allows more memory for the network. When the traffic subsides, memory is given back to the system. The decoupling of data structures from mbufs allows for better usage of memory for small structures.

3.7.5. Left To Do in the Network

As of this writing, the network is completely dynamic. If the system has sufficient memory, any number of network transactions are handled without modifying the kernel. The job of decoupling the network related structures from mbufs is nearly complete. Most protocol control blocks, socket structures, route table entries, imp host table entries, fragmentation reassembly queue headers, and interface information are no longer stored in mbufs. Socket options, arp table entries, and a few protocol control blocks are yet to be dynamically allocated.

3.8. Devices

A device driver's probe routine accomplishes most of the work in initializing the device. Traditionally, this routine contains static arrays for data such as controller block structures and device specific structures. Additionally, the probe routine initializes global variables used in other driver routines such as interrupt service routines and strategy code.

The prototype system consolidates static structures and arrays under dynamically allocated equivalents. This means no data and no bss segments are used for nonexistent devices. The DEQNA Ethernet¹⁰ driver demonstrates this case. The DEQNA previously held 1680 bytes of bss data per interface. Now a 4-byte pointer is used and the 1680-byte `softc` structure is allocated at probe time. Structures statically defined in header and data files are now defined as pointers validated when a controller is found and initialized at auto-configuration time. Similar savings are obtained with disk drivers (`uda.c` recovers 3K of data + bss) and the workstation screen driver (`qd.c`, over 1 K of data + bss).

Some device drivers require other resources called map registers. Map registers allow a device in I/O space to find physical main memory used for DMA operations. The entire resource map is allocated when the device is first initialized, rather than allocating the map at boot time (which is done in a static system).

3.9. Namei Pathname Buffers

Approximately 20% of the ULTRIX system calls take pathnames resulting in file system name translation (`namei`). Previously, these pathnames were copied into buffer space taken from the buffer cache. After `namei` finishes parsing the pathname, the buffer space was placed on a list of free pathname buffers. This buffer space is never returned to the buffer cache.

The prototype system uses `malloc` to obtain temporary space for path names. This space returns to the free memory pool rather than keeping a free pathname list.

3.10. Arbitrary Limits in Select

The `select` system call allows for synchronous I/O multiplexing. The call takes a mask of file descriptors to examine for reading, writing, or if an exceptional condition is pending (`SIGURG`). In order to handle an arbitrary number of file descriptors, `select` dynamically allocates the space needed for each of the masks. Since `select` is an expensive system call, this allocation is done only when more than 96 file descriptors are passed. This is in anticipation of dynamically allocating the pointers to file structures present in the `uarea`.

3.11. System Page Table Cleanup

There are two modifications in the handling of the system page table. First, the page table moves from the *data segment* to the *bss segment*. This involves a small program to take the page table specification and convert it into *external common declarations*. These declarations allow the support of very large memory systems with a small total text + data

¹⁰ Ethernet is a trademark of Xerox Corporation.

size. For example, on a 512 MB VAX processor, the operating system's total text + data is about 500K while bss needs 3 MB (2.7 MB is reserved for the system page table). If the system page table is not moved, text + data becomes about 3.2 MB.

Second, with the memory allocator having a large virtual address space, it is useful to take virtual address space for device drivers. On static systems, each device driver reserves a segment of the system page table. For instance, a disk driver adds 256 entries to perform I/O. The system page table grows at an alarming rate. A routine called *get_va_pte* reserves a virtual address range and assigns it to a set of page table entries. The allocator therefore acts as a resource for virtual address space and page table space for system device drivers. This reservation allocates no memory but uses page table entries to map to *other system memory* (e. g., I/O space) to perform the I/O.¹¹ This also helps control the growth of the system page table and aids in its management.

3.12. Kernel Profiling

One of the primary drawbacks of a kernel compiled with profiling is the allocation of a large buffer to store profiling data. The prototype kernel no longer allocates the 512 KB buffer needed for profiling at boot time. Instead, the space is allocated when the *kgmon* program enables profiling. Unfortunately, no simple way exists to give the memory back. Data can be passed using a system call rather than reading and writing */dev/mem*. This allows the return of memory used by profiling.

3.13. Kmemu – Memory Usage Reporting

A user level program for reporting memory usage is included. It helps in understanding and debugging the dynamic kernel. Figure 4 is a sample output from "kmemu" from a prototype kernel.

4. PERFORMANCE AND MEMORY UTILIZATION

Performance and memory utilization are important topics in the prototype. Structure element calculation shifts from index computation to pointer traversal. If this work is to have any bearing on future products, performance must be comparable at both the system call level and the overall system. Further, memory use must be examined and algorithms tuned when appropriate.

Performance has been measured on a MicroVax II with 9 MB of physical memory. The prototype kernel is contrasted against a version of Ultrix 2.0 (on which the prototype is based). The malloc kernel allows the buffer cache to grow to half of physical memory.

4.1. System Call Performance

The system calls are broken into three groups: process, file system, and network. Each bar chart measures individual system call performance for those groups. Shown is real time (the amount of time the system call test program took to complete) and system time (the amount of time the program spent executing in the kernel). The bars on the chart are not cumulative. For instance, for the malloc kernel, *execve* shows an improvement of about two percent in real time performance and an improvement of about five percent in system performance.

Figure 5 depicts the performance impact from allocating process related structures dynamically. Process structure manipulation increases the efficiency of process creation and deletion. *Execve* improves because the prototype allows program data space to be fetched from the buffer cache while the static system reads the data space from disk for each execution of the image. *Kill* shows no degradation even though the pid hash no longer exists.

¹¹ I/O pages must be treated specially so accesses to */dev/mem* do not crash the system.


```

1 KB in bucket 4 ( 64 - 16 byte chunks) with 63 on freelist
14 KB in bucket 6 ( 224 - 64 byte chunks) with 17 on freelist
88 KB in bucket 7 ( 704 - 128 byte chunks) with 51 on freelist
91 KB in bucket 8 ( 364 - 256 byte chunks) with 1 on freelist
19 KB in bucket 9 ( 38 - 512 byte chunks) with 2 on freelist
14 KB in bucket 10 ( 14 - 1024 byte chunks) with 8 on freelist
18 KB in bucket 11 ( 9 - 2048 byte chunks) with 5 on freelist
44 KB in bucket 12 ( 11 - 4096 byte chunks) with 0 on freelist
4120 KB in bucket 13 ( 515 - 8192 byte chunks) with 1 on freelist
80 KB in bucket 14 ( 5 - 16384 byte chunks) with 0 on freelist
64 KB in bucket 16 ( 1 - 65536 byte chunks) with 0 on freelist
28 KB in bucket 17 ( 1 - 131072 byte chunks) with 0 on freelist
256 KB in bucket 18 ( 1 - 262144 byte chunks) with 0 on freelist
512 KB in bucket 19 ( 1 - 524288 byte chunks) with 0 on freelist

```

Total Allocated = 5449 KB; Total Free = 35 KB

Memory Usage: Type and Number of Active Pieces

MBUF	2	DEVBUF	2	SOCKET	58	PCB	101
RTABLE	2	IFADDR	2	SONAME	2	SUPERBLK	4
TEMP	1	SELECT	29	MOUNT	18	CRED	23
RPC	10	RMAP	6	FLOCK	2	QUOTA	2
NCH	1	TTY	1	CALLOUT	1	CHROUT	1
PROC	34	FILE	133	SELBITS	1	BUFHEAD	514
BUFSP	513	GNODE	318				

FIGURE 4 – Kmemu Output

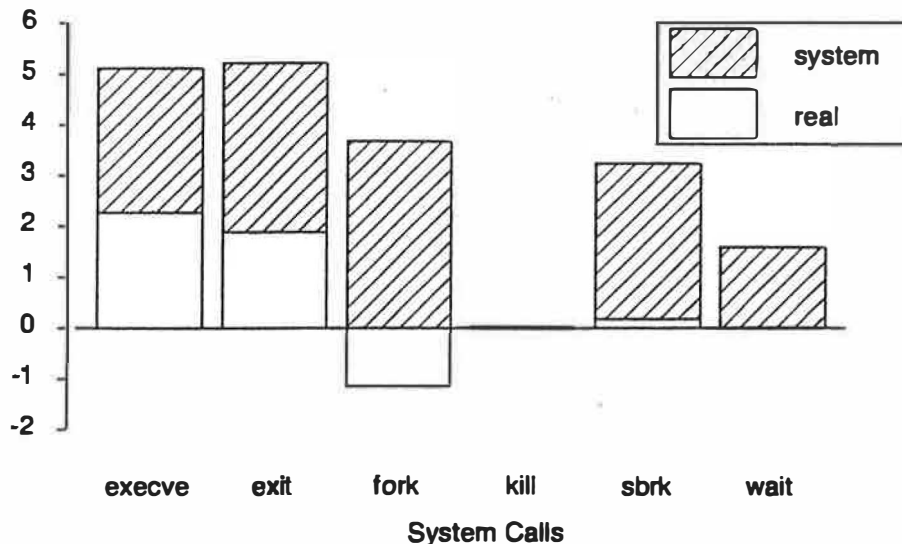


FIGURE 5 – Process Percentage Improvement

The performance of file system system calls is shown in Figure 6. Dynamically allocating file structures improves the performance of the *open* and *close* system calls. *Read* and *write* show a slight degradation, attributable to dynamically allocating buffer space during the system calls.

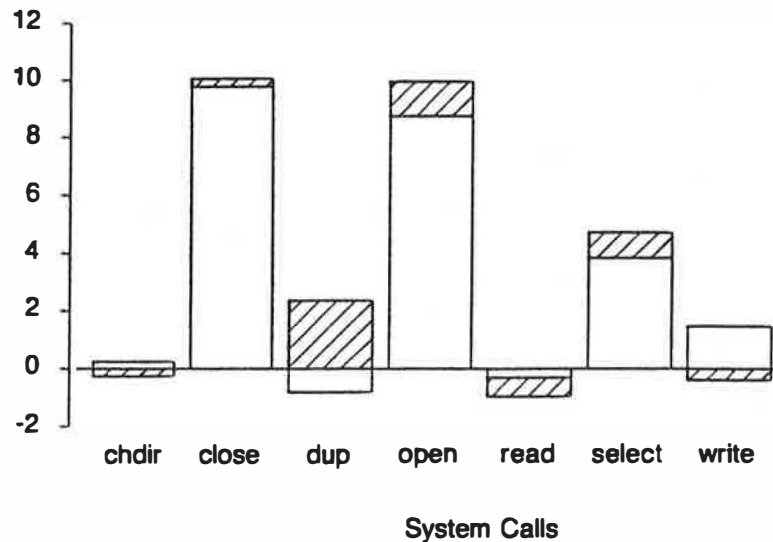


FIGURE 6 – File System Percentage Improvement

Figure 7 show system call performance for the network. This chart not only contrasts using the allocator with the old method of mbuf allocation, but also contrasts the 4.3BSD network code (present in the prototype) with the 4.2BSD network code. Dynamically allocating sockets and mbufs have not affected performance. This is expected since the prototype system provides memory for network resources from the allocator while the static system provides memory from the mbuf pool. It should be noted the *send* system call is slower due to changes in the 4.3BSD networking code.

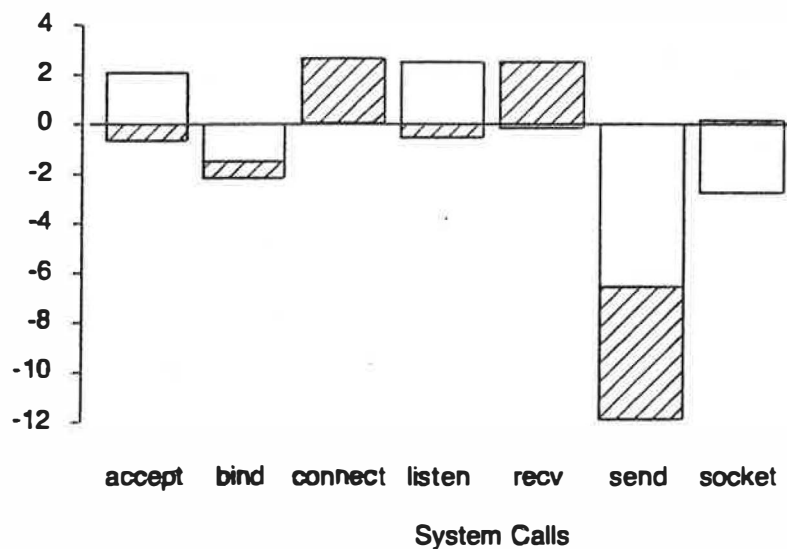


FIGURE 7 – Network Percentage Improvement

4.2. Overall System Performance

System call performance improves with dynamic allocation of kernel data structures. However, system call performance does not measure how the overall system responds. Understanding how the new kernel memory allocator interacts with the virtual memory system in a "production" environment is important.

System performance has been measured by simultaneously executing 1, 2, 4, 8, 16, and 32 scripts. These scripts introduce a job mix containing commands typically executed by a code developer.¹² Figure 8 depicts script throughput on the dynamic and static systems. The two curves are closely coupled until 32 scripts are executed in parallel. This degradation can be attributed to paging activity on the dynamic system.

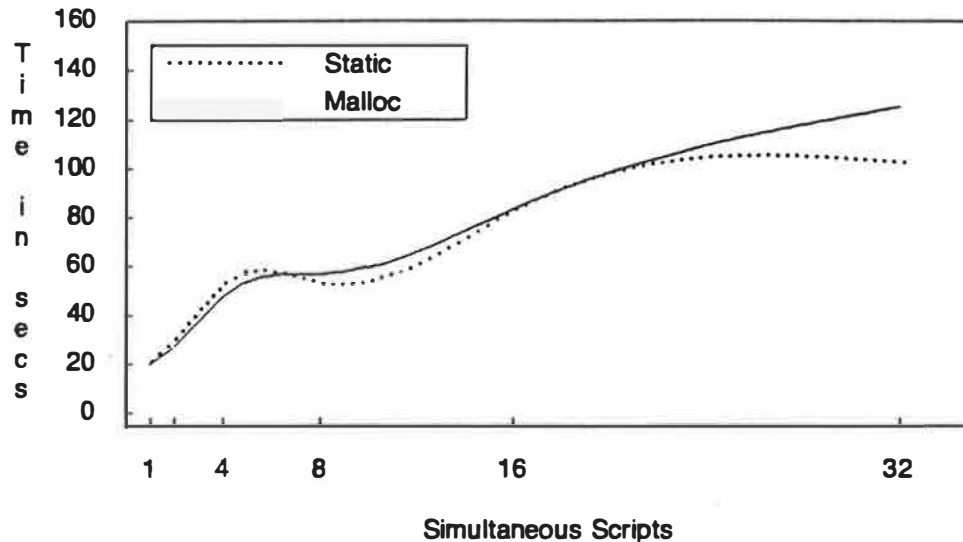


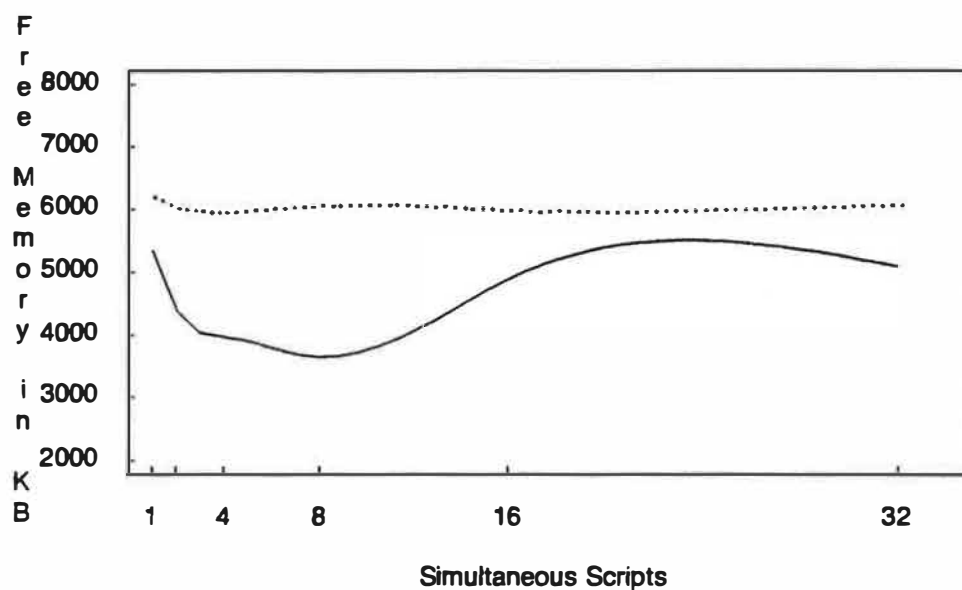
FIGURE 8 – Script Throughput

4.3. Memory Utilization

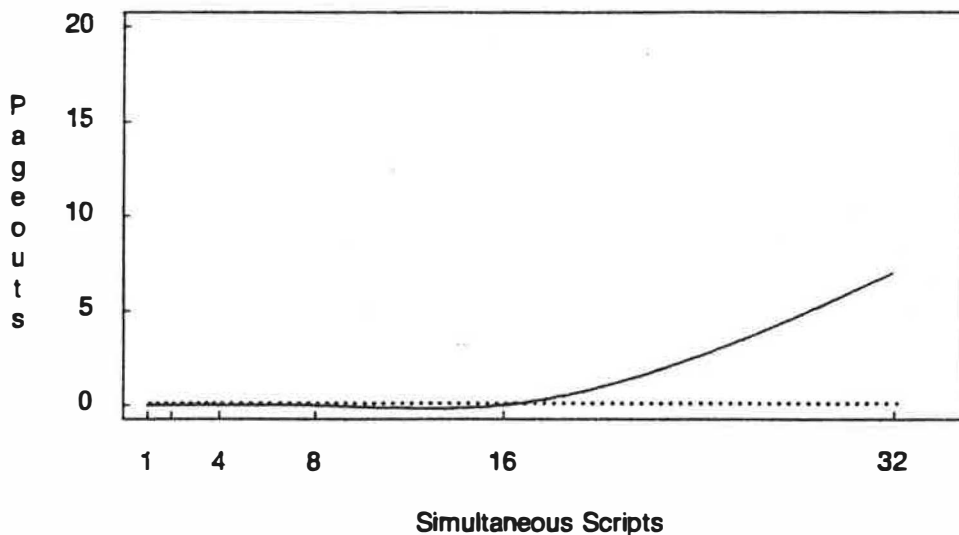
One of the primary goals of this prototype is better utilization of physical memory. The static system "allocates" 292 process structures, 26 file structures, 36 text structures, and 64 mount structures. This consumes approximately 57 KB. On a multi-user MicroVax-II with 7 X-windows and various other processes active, the prototype system allocates 35 process structures, 132 file structures, 23 text structures, and 12 mount structures. This consumes approximately 24 KB. Since allocation sizes are rounded up to the next power of two, the dynamic system wastes approximately 9 KB of the 24 KB. Judicious use of unions and general cleaning up of kernel structures should eliminate most of the wasted space. The static system allocates 192 gnodes, while the prototype allows 350 gnodes to be created. Finally, the static system has 172 buffer headers and 900 KB of buffer space. The dynamic system permits the allocation of 512 buffer headers and 4.5 MB of buffer space.

The allocation and return of buffer space in the prototype system plays heavily in system responsiveness. Figure 9 contrasts memory utilization between the prototype and the static system. There is an inverse proportion between the amount of memory consumed by the kernel allocator and memory available to user processes. When the first script starts, free memory is plentiful. Therefore the buffer cache is allowed to grow. As more scripts execute in parallel, the amount for free memory decreases while the size of the buffer cache remains

¹² The contents of the script is meaningless. The purpose of the script is to induce a reproducible load.

**FIGURE 9 – Free Memory**

constant. When sixteen scripts execute, free memory is reduced to the threshold where the buffer cache must be pruned. This returns memory to the free memory pool. The size of the buffer cache achieves steady state when paging is started, hence the amount of free memory is constant between 16 and 32 simultaneous scripts. Figure 10 shows paging activity during the execution of the scripts. Controlling the minimum and maximum sizes of allocated, cached, kernel data structures can control the threshold at which paging begins.

**FIGURE 11 – Page Outs**

5. LESSONS AND FUTURES

In general, allocating structures as needed is good. Performance is good. A few operations are slower, most are comparable, and some are faster. Overall, memory resources are better managed.

The dynamic allocation of the buffer cache can interfere with the virtual memory subsystem. Allocation for the buffer cache needs to be closely monitored to avoid thrashing due to of paging. Research needs to continue to find optimal algorithms for buffer cache and virtual memory interaction.

Because of the very dynamic nature of the structures, a generalized routine is likewise needed to fetch system data structures. For instance, *ps* has difficulty traversing the linked list of processes. A generalized system call to get and set system parameters is needed. Coupling these two system calls will allow for real time monitoring and tuning of systems.

By its nature, the network is very dynamic. The first goal is to allow the network to handle peak activity as efficiently as possible. No less important is to remove inappropriate uses of mbufs from the system. The memory allocator supports both of these goals.

In restructuring the code, the groundwork is laid to make device drivers "loadable". Indeed, if all global variables are confined to one dynamic structure, the driver prologue tables defined at configuration time are stubs filled in when the device is first accessed. When the kernel probes a device, the stub is vectored into routines specific to that device. If the device does not exist, the routines are never pulled in.

A kernel containing loadable device drivers and dynamically allocated kernel structures will run on every system. System administrators will never have to reconfigure or tune and operating system for a specific machine.

6. REFERENCES AND ACKNOWLEDGEMENTS

There are many people deserving mention. This paper would not have been possible without Kirk McKusick and Mike Karels' kernel memory allocator. We thank the score of people who have critiqued this paper. Finally, we thank our fellow ULTRIX engineers who have supported us in this endeavor.

References

Ritch1984a.

D. M. Ritchie, "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1577-1593, 1984.

Ritch1978a.

D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1905-1929, 1978.

Thomp1978a.

K. Thompson, "UNIX Implementation," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1931-1946, 1978.

McKusl1988a.

M. K. McKusick and M. J. Karels, "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel," *USENIX Conference Proceedings*, Summer, 1988.

Leffl1983a.

S. J. Leffler, W. N. Joy, and R. S. Fabry, *4.2BSD Networking Implementation Notes*, July, 1983.

Stellix UNIX for a Graphics Supercomputer

Thomas J. Teixeira and Robert F. Gurwitz
Stellar Computer Inc.
85 Wells Avenue
Newton, MA 02159

ABSTRACT

The Stellix™ operating system has been specifically designed for the Stellar™ Graphics Supercomputer Model GS1000™, the first of a new class of systems that tightly integrates the processing capabilities of mini-supercomputers with high performance 3-D graphics in a single-user package. Stellix is a multithreaded version of UNIX® System V Release 3.0.

To fully exploit the multistream architecture of the GS1000, Stellix provides a lightweight process abstraction, known as *threads* for medium-grained parallel programming. The operating system also supports special hardware mechanisms for fine-grained concurrency. Our FORTRAN and C compilers can automatically detect parallelism and generate code that uses these hardware concurrency mechanisms.

In this paper, we give an overview of the Stellix operating system, including the software environment provided to the user. We focus both on operating system support for the multistream architecture of the GS1000 and on the parallel programming support features for both medium- and fine-grain concurrency.

1. Introduction

Stellar Computer Inc. has recently introduced its first product, the Stellar Graphics Supercomputer Model GS1000. This machine represents a new class of systems that offer the integer and floating point performance features of mini-supercomputers combined with tightly integrated high-performance 3-D graphics. The GS1000 is designed as a single-user system, an evolution of the bitmapped single-user graphics workstation that became prevalent in the early '80's. With its increased CPU power and 3-D graphics capabilities, the new graphics supercomputer will find its application in areas such as finite-element analysis, computational fluid dynamics and molecular modeling, all of which will benefit greatly from the high levels of interactivity and real-time response that the Stellar system provides.

The operating system for the graphics supercomputer has special performance and functionality requirements for supporting these demanding applications. The

STELLAR, STELLIX, GS1000, SPMP, SPC/386, and DataPath are trademarks of Stellar Computer Inc.

UNIX is a registered trademark of AT&T.

NFS was created and developed by and is a trademark of Sun Microsystems, Inc.

MS-DOS is a registered trademark of Microsoft Corporation.

X Window System is a trademark of MIT.

architecture of the GS1000 makes available a number of processing resources that can be used in parallel, causing increased demands on the operating system to manage these resources without imposing excessive overhead. These requirements have affected the design of the operating system in a number of areas, especially the process structure and support for parallel programming. In this paper, we will examine each of these areas and describe how the requirements imposed by the graphics supercomputer affect the operating system design and implementation.

1.1 Hardware Architecture

The Stellar GS1000 Architecture centers around a novel CPU organization called SPMP,TM for Synchronous Pipelined Multi-Processor (Figure 1). The SPMP CPU simultaneously executes four instruction streams in a single, 12-stage 50 nanosecond pipeline to achieve a 20-25 MIPS integer execution rate. The system can support between 16 and 128 MB of main memory. The four instruction streams in the SPMP CPU share a single 1 MB cache, which obviates the need for explicit cache synchronization between streams. The system also has a 16K entry translation lookaside buffer (TLB) for virtual memory management.

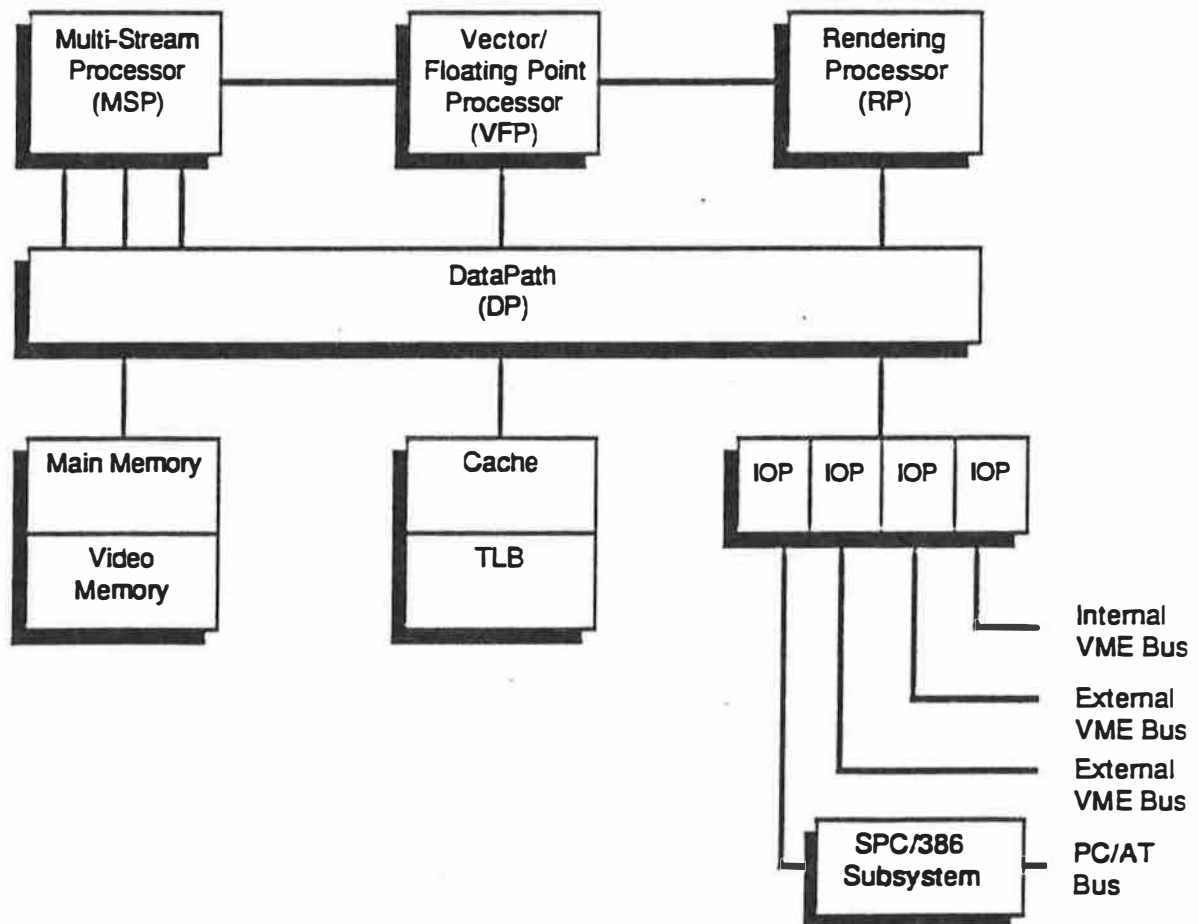


Figure 1. Stellar Graphics Supercomputer Architecture

The SPMP CPU also controls a number of other independent functional units which can be accessed for both compute and graphics processing. These include: 1) a single vector floating point unit with a peak capacity of 40 MFLOPS, used for numeric

computation and graphics transformations, 2) a special purpose rendering processor which consists of 16 SIMD processors running at 20 MIPS, used to render 16 pixels in parallel during each cycle, and 3) up to four independent I/O processors each with a sustained bus transfer rate of 16 MB/second, used to connect industry standard VME and PC-AT compatible busses. An 80386-based service processor, the SPC/386,TM subsystem is used for system bootstrap and diagnostics, connection of serial I/O devices on the PC-AT bus, and as a platform for running MS-DOS[®] applications during normal system operation.

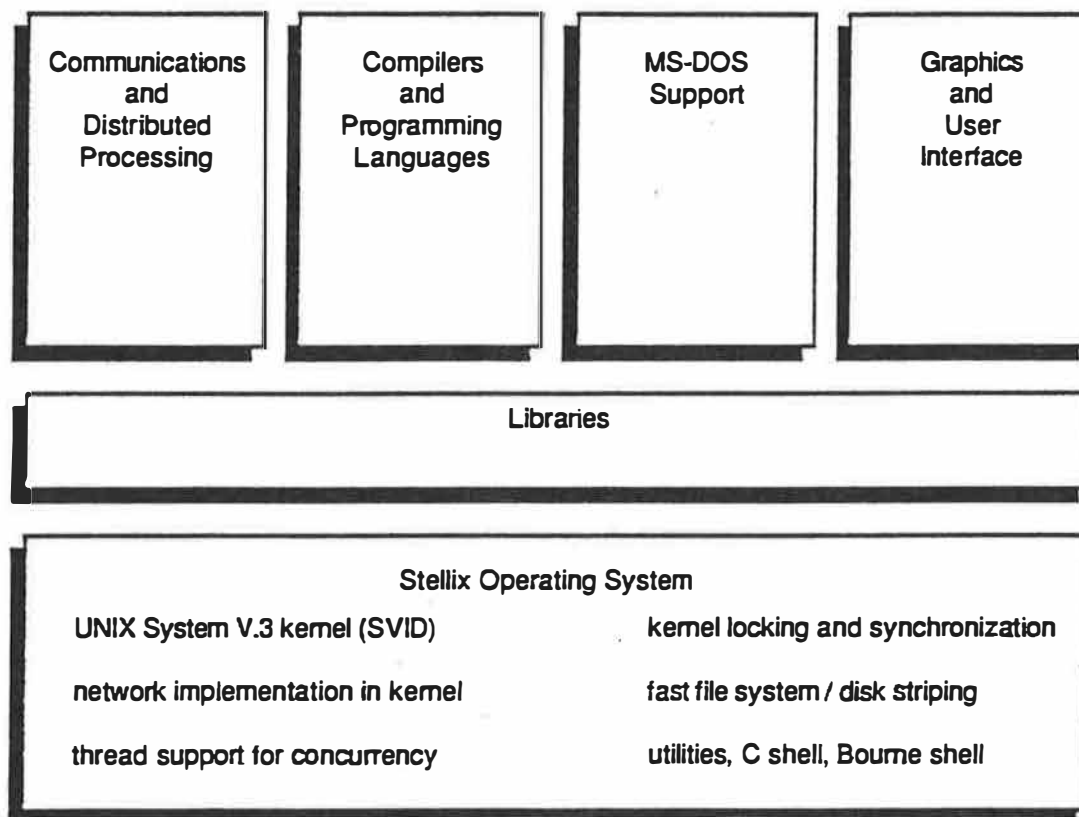


Figure 2. Stellar Software Architecture

The core of the processor is the DataPathTM architecture, which provides the large bandwidth for data transfer among all the functional units. This enables the high levels of integration for functional units and execution streams needed for high performance numerical processing and 3-D graphics rendering. Some examples of the bandwidths which the DataPath provides are 1280 MB/second to the cache, 480 MB/second bandwidth to the integer and vector units, 320 MB/second bandwidth to the memory, and 640 MB/second bandwidth to the renderer.

1.2 Programming Environment and Software Structure

The programming environment on the Stellar GS1000 is built on the UNIX System V Release 3.0 operating system, enhanced with many Berkeley UNIX (4.3BSD) features (Figure 2). We chose System V.3 UNIX as the base for Stellix because it provides a widely adopted standard that has a rich utility set and the flexibility to accommodate the functional extensions that the graphics supercomputer requires. To System V.3 UNIX we have added Berkeley symbolic links, long filenames, signal handling, and sockets. We

have also picked up many 4.3BSD utilities. The device driver and internal network protocol structure have also been largely adapted from 4.3BSD.

The communications software includes a complete implementation of the Internet protocol suite (including Berkeley networking utilities), and the Sun Network Filesystem (NFS). Both FORTRAN and C compilers support automatic vectorization and parallelization and sophisticated global optimization and instruction scheduling algorithms. The graphics software support is based on the X Window System™ (version 11) and PHIGS+. In addition, the SPC/386 service processor can be used to run off-the-shelf MS-DOS applications in an X window.

2. Multithreading Stellix

The multistream nature of the GS1000 architecture requires the operating system to be capable of running all four execution streams with minimal overhead. The most general way of doing this under UNIX is to be able to execute the operating system kernel on all streams in parallel, and provide the synchronization and locking required to protect critical OS data structures and machine resources. Thus, processes (and threads, Stellix's lightweight process abstraction) can be scheduled to execute on any available stream and are able to make system calls, take interrupts, and handle page faults and TLB misses.

Running the operating system on a multiprocessor changes some of the fundamental assumptions about the way the UNIX operating system works. Multithreading implies that more than one process can be running kernel code at any time. Traditional UNIX synchronization and locking, using the *sleep/wakeup* mechanism, assumes only one thread of execution in the kernel at any time and fails in a multithreaded kernel. On a uniprocessor, interrupts that could violate critical code sections can be masked-off during these critical sections by manipulating the processor interrupt priority. By removing the assumption of a single thread of execution in the kernel, we must introduce explicit locking mechanisms to prevent corruption of system data structures during interrupts. The synchronization and locking mechanisms which we designed for Stellix are described below.

2.1 Kernel Locking Primitives

The Stellix kernel synchronization mechanisms were designed with a number of goals in mind. First, we implemented both spin locks and sleep locks to minimize CPU overhead and lock contention. Spin locks simply perform a repeated test of a flag variable to determine whether the lock is set or not. To implement spin locks, the GS1000 instruction set provides a set of atomic instructions (fetch-and-add, fetch-and-subtract, bit set, bit clear, bit test, and compare-and-store). Spin locks are used for low level locks that are guaranteed not to be held for long periods of time. Sleep locks are similar to the traditional UNIX *sleep/wakeup* mechanism, but are correct for a multiprocessor: if the lock is held by another process, the calling process is dismissed; when the lock is released, a process waiting for the lock is awakened. Just as with *sleep/wakeup*, it is important that the operating system be able to abort waiting for certain sleep locks, particularly in response to a signal. In particular, the existing UNIX model using non-local goto's must be preserved.

Second, to allow both efficient protection of shared resources and efficient resource allocation, Stellix provides both mutual exclusion and producer/consumer synchronization. Also, in order to improve the amount of parallelism in the kernel, readers/writers locks are supported. This allows multiple readers to share data that they access in a read-only fashion, while protecting the data from write modification.

Finally, we decided to adopt a locking protocol to avoid deadlocks by helping to ensure that locks are held and released in the correct order. The protocol works by assigning levels to each lock, determined by a partial-ordering of locks in the system that need to be held at the same time as other locks. The locking operations are then called with these lock levels as parameters and debugging code verifies that the locks have been locked in the correct order. Otherwise, a debugging assertion fails and the system panics.

The basic locking primitives used in Stellix are integer semaphores (with interface routines as described by Bach and Buroff [1]) and readers/writers locks. In addition, we defined *events* which are used for signalling the occurrence of an event (the basic operations are read the event count, advance the event count, and await the event count reaching a specified value)[3]. We also added *resources*, which are like semaphores, but which may be adjusted by an arbitrary value (not just plus or minus one). Spin locks are used for the lowest level locks that require minimal checking. Spin locks also disable interrupts and reenables them at the end of the critical section when the lock is released.

All of the sleep locks take flags similar to *sleep/wakeup*, which specify a scheduling dispatch priority that also determines whether the process may be interrupted by a signal, whether or not to catch the signal, a flag to indicate that the value of the lock should only be tested without sleeping, and a flag to indicate that the caller should busy wait for the lock rather than sleep for it. The non-sleep variants of the locking operations are important for acquiring locks in interrupt handlers and acquiring locks out of order. This is necessary for some locks which cannot fit into the partial-ordering of levels, such as two locks at the same level which need to be held at the same time.

2.2 Other Multiprocessor Considerations

Another change in the operating system necessitated by multithreading is in context switching. Early UNIX implementations performed a full context switch to a separate scheduler process to run the code which selected the next process to run, rather than running the scheduling code in the current process and switching directly to the new process. In 4.0BSD, Joy observed that the scheduling code could always be run safely in an idle or defunct process on a uniprocessor, since the only code that could run during the dispatch was interrupt handlers[2]. This is not the case on a multiprocessor. For example, in the System V.3 context switch routine a check is made to see whether the current process is exiting and releases its memory if it is. On a multiprocessor, another processor could be allocating the memory that was being released and overwrite the kernel stack being used by some other processor.

To avoid these problems, the Stellix kernel reserves a special stack for each processor that is used when the processor is idle. When the current process enters the sleep or zombie state, the system does a mini-context switch to this idle "process," which in fact is only a stack switch, since the user and process structures are not referred to in these states and stack variables are not referred to.

Interrupts are another area where the traditional UNIX approach of manipulating processor interrupt priority does not work in a multiprocessor. Interrupt handlers require replacement of *sp1* calls with explicit locks to ensure exclusive access to data structures which are shared between the "call side" and the "interrupt side" of device drivers. However, special rules must be observed for these locks to avoid deadlock. First, interrupt handlers must not sleep when acquiring these locks since they may not be running in the context of any process. Therefore, only the test or busy wait variants of locking operations are used by interrupt handlers. Any processor that acquires a lock which is also acquired by an interrupt handler must not allow interrupts after acquiring the lock. Otherwise, an

interrupt handler could deadlock with a "call side" routine running on the same interrupt handler. Also, since putting a process to sleep has a side effect of enabling interrupts, a processor should not sleep after acquiring a lock used by an interrupt handler. The way Stellix deals with these problems is by defining a boundary level between locks which must inhibit interrupts and other locks. The locking primitives then automatically inhibit interrupts after acquiring a lock at or below this level.

2.3 Locking Structure

The design of the Stellix multithreaded kernel started with a base System V.3 kernel. Once the basic locking primitives were defined, the primitives were implemented and the base kernel was isolated. The base kernel of the operating system is responsible for implementing the process abstraction and scheduling these processes for execution on processors. Although System V.3 does not have a well defined kernel, the operations implemented by the System V.3 kernel include sleep, wakeup, sched, disp, and swtch. The Stellix kernel implements these operations, although sleep and wakeup have been replaced by the new synchronization primitives. The base kernel is single-threaded by a single spin-lock.

System calls are single-threaded at the top level to prevent corruption of shared data structures between threads of a process (see below) by use of a per-process system call lock. This lock is held until the affected data structures can be locked (for example, the file table). Device drivers require a "call side" lock and an "interrupt side" lock to single thread execution of the main body of driver code and the interrupt handlers, as described above.

The data structures in the rest of the operating system are protected by a number of other locks. To do this, the system was divided into its major subsystems: file system, paging system, network code, device drivers, and so forth. In each area, the data structures were examined and the appropriate locking and synchronization was inserted. A balance had to be achieved between the level of parallelism allowed in the operating system and the number of locks and concomitant overhead imposed by the locks, as well as potential for lock contention. In most areas, we tried to minimize the number of locks, and keep the locking code as simple as possible, even at the expense of some parallelism. We use readers/writers locks where possible to maximize parallelism, and avoid using locks where possible by using atomic instructions, such as compare-and-store sequences to insert elements on singly linked lists.

In practice we found that the original locking primitives (semaphore and readers/writers locks) needed to be augmented. For example, using semaphores for purposes other than mutual exclusion caused problems with the locking protocol level checking. The reason for this is that the level checking code assumed each semaphore *P* operation to be paired with a *V* operation in the same process. However, for certain uses where the semaphore protects a resource that is contended for by multiple processes, this assumption breaks down. One example of this was locking buffers in the disk block cache where a buffer may be locked by one process starting an asynchronous I/O operation, but the buffer is unlocked by *iodone* called from an interrupt handler. Also, it is possible to perform *P* operations on more than one semaphore at a given level without an intervening *V* operation, but the number of unpaired *P* operations is severely limited. In this case, the lock level can be set to zero to avoid the check, but it is more natural to use the event primitive in these cases. Though we found the locking protocol to be useful in finding potential deadlock situations, there were cases where the debugging code was more of a hindrance than a help and lock violations occurred which were more an artifact of the lock checking than a useful indication of deadlock.

3. Parallel Programming Support

UNIX processes are inappropriate for implementing fine grained concurrency because processes do not share the right context. Stellix has replaced the classical UNIX notion of processes with *threads* and *processes*. A thread is a lightweight process that is essentially a flow of execution within an address space and associated operating system context. For example, all threads of a process share an open file table so that files opened by one thread can be accessed by all other threads of the process.

However, threads have separate kernel stacks and some independent kernel state. This allows threads to take page faults and make system calls independently and makes threads suitable for many instances of programmer explicit multitasking in addition to compiler generated concurrency.

Threads are created explicitly by a *tfork* system call. Creating a thread is conceptually more complex than creating a process: since the old and new thread share an address space, a portion of the virtual address space must be allocated for this new thread to use as a stack. The time required for *tfork* is roughly comparable to that required for *fork*, although a newly forked process will immediately encounter copy-on-write faults which will not occur in a newly created thread. Threads are destroyed using a *textit* call.

Stellix does not implement a distributed notion of threads or processes. That is, there is no attempt to extend the operations of process creation or synchronization, including *fork*, *exec* or *kill* to work between nodes over the network. Similarly, all threads in a process are constrained to run on a single node and the synchronization mechanisms used for threads are not necessarily efficient for a truly distributed application.

The implementation of threads partitions the major kernel data structures associated into two pieces — a per thread piece and a per process piece. Thus, Stellix has both a *proc* structure (per thread) and an *sproc* structure (per process) as well as a *user* (per thread) and *suser* (per process) structure. A per process lock is part of the shared user structure and is used to control access to other parts of the shared process state. Additional locking is not required since other processes do not modify process state of other processes (with the exception of very small amounts of state modified by the swap scheduler which operates under the basic kernel lock). As in traditional UNIX systems, the unshared user structure is mapped at the same virtual address space in all threads. The unshared data is accessed through pointers located in the unshared user structure. Although it would be possible to hide the specific location of fields in these data structures by judicious use of macros, we elected to change the references throughout the code. This made it more obvious where additional locking was required.

3.1 Fine-grain Concurrency Support

While threads provide an underlying mechanism for multiple processors to execute on behalf of a single user program in a single address space, forcing inter-thread synchronization to go through the operating system would not result in a mechanism suitable for fine-grained parallelism. Instead, most thread-to-thread synchronization is done using the same atomic operations on shared memory used by the kernel to implement multithreading.

While shared memory provides for much more efficient inter-thread synchronization than system calls, the overhead of additional memory references may still be significant in small, concurrent loops. The GS1000 implements four sets of two shared registers that can be used for synchronization or inter-thread communication in

concurrent programs. These concurrency registers are accessed by special instructions which implement load, store, fetch-and-add, and various forms of single bit test-and-set. Typically these registers may be used to hold the iteration count for a parallel DO-loop, or flag bits used to lock access to shared variables.

All atomic operations on concurrency registers execute in one instruction time, and do not need to reference memory. The corresponding atomic instructions which operate on memory may be delayed for cache misses. Writes to memory will also cause a delay if the next instruction in the stream needs to reference memory. In addition, atomic memory operations may interfere with other streams attempting to access the same line in the cache since the read and write of the memory must be atomic.

The GS1000 allows for more than one concurrent program to execute at any given time and includes hardware support for dynamically allocating one of the shared register sets to an instruction stream. Each block of code which may execute in parallel is associated with a *parallel region descriptor* (PRD) which contains the state of a concurrency register set and the address of that block of code. The concurrency register set associated with a PRD is virtual since a process can have any number of PRDs. However, at any instant in time, a stream can have at most one real concurrency register set assigned to it. The header for the code block initializes the PRD which is stored at a well known address. Other threads in the process which are not actively executing code are in an "idle loop", and repeatedly examine the PRD waiting for it to become active.

A thread enters a parallel region by executing an *epr* (enter parallel region) instruction. If other threads are executing within that region, the current thread attaches itself to the physical concurrency register set in use by the other threads. Otherwise a new physical concurrency register set is allocated for the virtual concurrency register set represented by the PRD and initialized. After executing all code in the block, each thread exits from the parallel region using an *lpr* instruction and returns from whence it came (either the idle loop or the main body of the program).

If a thread is preempted, the operating system will temporarily exit the parallel region on behalf of the current thread. If the number of active threads becomes zero, state information is stored back into the PRD and the concurrency register set is made available for other threads. The operating system will re-enter the parallel region on behalf of the thread the next time it is scheduled for execution.

A program can have nested parallel regions. This can arise if a program calls a subroutine that uses concurrency while the program itself is executing within a parallel region. In this case, the subroutine must temporarily suspend executing within its current parallel region before establishing a new parallel region. Depending on the complexity of the idle loop, other threads may or may not be aware of this new parallel region: if no other threads are available, the code block is executed by only one thread instead of many.

3.2 Stagnation and Deadlock Detection

Threads are the fundamental scheduling unit in Stellar, not processes. Therefore, it is entirely possible (and indeed common) for some threads of a process to be executing while others are blocked or simply preempted by higher priority threads. This raises the possibility of stagnation where one or more threads are waiting for results that must be computed by a thread which is not currently executing. The GS1000 is able to detect this and generate a processor fault.

Executing special instruction sequences used for synchronization will classify a thread as "waiting". The machine state includes an indication of which process each thread belongs to. If all threads in a process are waiting at the same time, the hardware generates a stagnation fault.

A stagnation fault may indicate that a program has a coding error, and all threads are waiting for an event which will never happen. More commonly, a stagnation fault indicates that some other thread in the process should be scheduled for execution. It is not possible to unambiguously determine the underlying cause of a stagnation fault without detailed knowledge of the program structure. If all the threads of a process are currently executing, the stagnation fault must be due to a program error. However, the state of the processors at the time of the fault is not available, and some other thread of the process may have begun execution between the time the fault occurred and the time the fault handler began executing. Instead, Stellix uses a simple heuristic of estimating the frequency with which a process gets stagnation faults and declaring a deadlock fault if this frequency exceeds a preset threshold. Otherwise, Stellix simply preempts the current thread and chooses a new thread for execution. If the new thread is part of the same process, the stagnation may be resolved by executing this thread; otherwise, the remaining threads of the process will continue to get stagnation faults and be replaced by threads in other processes which are able to make progress.

More important than detecting stagnation, Stellix includes a mechanism to avoid stagnation where possible. The typical parallel block of code is a DO-loop, which may or may not have a data dependency between iterations of the loop. Suppose a loop begins executing using four threads, and all four threads are executing on streams when one stream is preempted. If the loop contains a data dependency, some other thread will block waiting for data computed by the preempted thread. A third thread may then become blocked waiting for data computed by the first waiting thread and a stagnation will occur. Eventually the operating system will schedule the thread that had originally been preempted. This thread will complete the computation for one loop iteration, begin execution of a new loop iteration, and then it will block waiting for input from one of the other threads. Although the computation will eventually complete, it will be extremely inefficient as the threads "thrash" while they compete for too few execution streams.

Part of the problem is that after the initial preemption, each thread will encounter a stagnation fault while it is in the middle of a loop iteration and, consequently, will contain the state essential to the proper execution of the program. However, at the end of each loop iteration, the thread has no state and could drop out of the computation at that point, allowing one of the waiting threads to proceed. At the point, the number of available streams would be in balance with the number of active threads and execution would proceed with no further stagnation faults.

In loops with no data dependencies, the problem is somewhat less severe since each loop iteration could be executed without waiting for the preempted thread. However, at the bottom of the loop, it is generally necessary to wait for all the threads that have entered a parallel region to exit that region and stagnation will occur during this wait. Again, the threads waiting at that point have no essential state and could quietly drop out of the computation to avoid further stagnation.

Stellix supports this method of balancing the number of active threads in a process against the number of available streams by maintaining a special variable in a shared memory area accessed by both the kernel and user programs. This variable, CMIT (for *CPU's minus threads*) is zero if all "active" threads are currently executing. The operating system will decrement this variable whenever it preempts a thread, and

increments this variable when the thread is next scheduled for execution. The user program in turn will increment this variable when a thread returns to the idle loop and decrement it when a thread leaves the idle loop and enters a parallel region.

At the bottom of each loop iteration, the user program can inspect this variable: if the value is negative, there are fewer streams than threads and this thread should leave the parallel region and return to the idle loop. In turn, the idle loop needs to examine this variable in addition to waiting for a parallel region to become valid: if the value is less than or equal to zero, the thread should remain in the idle loop since entering the parallel region could result in stagnation traps. The latter check is necessary primarily to prevent the thread which just returned to the idle loop from reentering the parallel region.

4. Conclusion

The key to high performance in the Stellar GS1000 Graphics Supercomputer is effective utilization of the multiple functional processing units of the hardware. This requires efficient management of these processing resources by the operating system, as well as mechanisms for making these resources available to the applications programmer with very low overhead. The Stellix operating system provides both functions by implementing a highly parallel multithreaded UNIX implementation, threads for medium-grain programmer-directed concurrency, support for hardware concurrency registers, and parallel region descriptors for compiler-directed fine-grain concurrency. This allows the Stellix operating system to take full advantage of the GS1000 multistream architecture for timesharing traditional UNIX processes, for programmed concurrency, or for automatically detected concurrency applied to existing "dusty deck" applications. The Stellix scheduler allows for efficient mixing of all three modes of processor use for maximum flexibility.

The GS1000 is the first example of a new class of computer, the Graphics Supercomputer. As such, the OS fulfills the high performance requirements for a large body of user-level system and application software, while providing support for architectural features not normally associated with workstation class machines: vector registers, multiprocessors, fine grain concurrency support in hardware, and very high performance 3-D graphics to name a few.

5. References

- [1] Bach, M.J. and S.J. Buroff, "Multiprocessor UNIX systems", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8 Part 2, pp. 1733-1749, October 1984.
- [2] Joy, W., *Comments on the Performance of UNIX on the VAX*, Computer Science Division, Electrical Engineering and Computer Science Department, University of California, Berkeley, 1980.
- [3] Reed, D.P. and R.K. Kanodia, "Synchronization with Eventcounts and Sequencers", *Communications of the ACM*, Vol. 22, No. 2, pp. 115-123, February 1979.

PROTOTEXT: UNIVERSAL TEXT DRIVERS

*V. Guruprasad
Kirkloskar Computer Services Ltd.
Bangalore 560 055 India*

ABSTRACT

Prototext provides Virtual Text Device interfaces to text oriented software, freeing applications from device and script problems. It is a semi-compiling language system for dealing with fonts, translational glyph composition, and almost any kind of printing or display device.

1. Perspectives:

The problem that I attempted to solve with Prototext is of maximally utilising the capabilities of output devices for depicting text. The elegance of the solution allows my Lisp interpreter to avail of "soft keyboards" and output text in Indian languages by simply piping with Prototext standalones on UNIX. "Ed" could be similarly used. Indian language syllable grammars (see ahead) are being developed by my colleagues, as another instance, on 25x80 character terminals via Prototext.

1.1. Programmability and Interfacing:

An interface must match the capabilities on the two sides. To optimise an interface, each side must be tuned. Flexibility of the interfacing requires that it be programmable separately from the applications, and is necessary both for switching across the peripherals and for tuning.

The first step is to encode selectors of device control algorithms within the application by a data file according to device capability. This is the approach of the UNIX device capability databases, viz the printcap and termcap of BSD versions and the terminfo of System V. The complexity of the device control problem however necessitates the provision for arithmetic in the format string in these databases, tending toward encoding of algorithms as well.

Applications tend to become omnipotent and omniscient with the use of these databases. When the usual entries do not suffice, proprietary extensions are made to the databases. For the problem discussed below, graphics devices are required, which meant the design of a rather large extension to the UNIX device capability databases, while the existing entries were of little use.

The control algorithms may instead themselves be encoded in the Prototext language, which reflects the symmetries involved in device control. Applications using the Prototext interface define their specific virtual devices. The Virtual Text Device (VTD) system of Prototext consists of device control solutions. Applications are then lighter to design, port, customise and use.

1.2. The Multi-Lingual Problem

Indian language scripts are generally phonetic; the syllables are graphically composed from elementary glyphs according to rules specific to the script. This means that a peripheral that can depict Indian scripts is capable of general graphics. Graphic composition in European scripts is trivially confined to concatenation; the accents are usually handled by enumeration in the font of all combinations of characters and accent signs. With Indian scripts, the composition is complicated by the innumerable partial consonants

and vowel signs, which are added or subtracted on all sides of the current syllable.

The multi-lingual problem contains the following aspects:

- 1.2.1. designing applications to be independent of the script; this requires the scripts to be transparent to the editing, a distinction between syllable level and character (vowel or consonant) level editing, and provision for either;
- 1.2.2. designing applications to be independent of I/O devices, and to have flexibility regarding the keyboard layout; this requires a way to configure the keyboard interpretation and the display and printing by data files;
- 1.2.3. designing a standard interpretation for every font (set of elementary glyphs) and every script (defining the syllable composition), and a standard format for fonts, to allow reconfiguration by data files;
- 1.2.4. designing an automatic transcription system to provide en-scription (e.g. translate "\alpha\" to a character code for the Greek symbol), de-scription (the inverse), and trans-scription (translate across scripts, e.g. "schul" to "school"); transcription is formalised by the syllable composition grammars of the scripts;
- 1.2.5. display operations like cursor positioning are no longer for fixed character cells; the "character" for cursor positioning would presumably be a syllable, which would vary in length; the "character" for insert or delete could be either the entire syllable to just the last entered consonant or vowel;
- 1.2.6. finally, provisions for downloading fonts and for graphic composition vary with the peripherals, requiring, amongst other things, device-specific arithmetic on font data.

1.3. Of Scripts, Character Sets and Fonts:

In the following, a script is defined to consist of:

- 1.3a.1. a list of names, termed characters,
- 1.3a.2. a mapping of the characters into an 8-bit code, termed character set,
- 1.3a.3. a graphic rendering for each character, termed the glyph, and
- 1.3a.4. rules prescribing graphic composition of the glyphs into syllables, termed the syllable grammar.

The character set is usually confined to the hexadecimal ranges 20 through 7f and A0 through FF. The range 0 through 1F is reserved for the Ascii control characters. The range 80 through 9F may be used for document controls. Prototext is, however, not dependent on the particular assignment of the character set.

A font denotes:

- 1.3b.1. the character cell size for a glyph,
- 1.3b.2. the glyphs themselves, and
- 1.3b.3. the calligraphy or style of rendition.

Prototext restricts the syllable grammar to translational additive operations, i.e., superposition with possible displacement and masking. Other operations, such as rotations, scaling and skewing, would require real-number computations.

1.4. Classification of Devices:

Most output devices are covered under the categories below. The character devices embody the notion of text: some form of pagination and scrolling, i.e. an inherent scheme for computing the next glyph position, are implemented in them.

1.4.1. Programmable Character Devices:

Most alphanumeric terminals and printers are PC devices, though the degree of programmability varies considerably. They hold font data and character set definitions internally. PC devices are totally text oriented: except for commands, every character received is mapped into a glyph. The fonts and the character sets (and character set pointers, viz G0 .. G3, though ANSI doesn't know them so) are often programmable. Syllable composition may be easily available on some (particularly the hardcopy ones), and the code for it may be downloaded on others (e.g. my original Fortune 2000 terminal kernel, version 1.0d).

1.4.2. Degenerate Character Devices:

These are defined by the collapse of the character cell along one dimension. They consequently take in graphic data together with text mode commands, such as line feed. Some expect their "characters" in rows, like rasters; many expect them as columns of dots. Their data formats vary, as does the "character cell" size.

1.4.3. Graphic Command Devices:

These are totally non-text, with no notion of scrolling at all. They expect commands addressing geometrical entities. Many systems provide graphics via interrupts or memory access by I/O calls.

2. PROTOTEXT:

I built Prototext as part of Totemtext, a totempole of text power, which includes Metatext, a script editor, and Orthotext transcriptions in Lisp. Orthotext satisfies requirement 1.2.4. above. Prototext contains compilers for fonts and for programs in the Prototext language, and interpreters in C.

2.1. Prototext Virtual Text Devices:

The Prototext Virtual Text Devices are defined as directories in a file tree, each containing the device program and, in the case of output graphic devices, the font file. The VTD are classified as:

INITABLE	if an init program is specified,
INFLOW	if an input device (else output),
GRAPHIC	if a font is used,
PLOTTING	if driving a Graphics Command Device,
DISPLAY	if a Virtual Text Terminal (VTT),
SHAREDPG	if using page owned by another VTD.

DISPLAY requires PLOTTING, which in turn requires GRAPHIC. SHAREDPG is applicable to GRAPHIC VTD. The VTD satisfy all the requirements in 1.2. above, except 1.2.4. and 1.2.5.

The Prototext font files contain a two byte header stating the character cell height and width in dots, and the character fonts as columns of dots, each column being of an integral number of bytes.

2.2. The C application interface:

The Prototext VTD interface to applications in C consists of the functions defined next. The standalones mentioned earlier use these.

```
/* (*vtd) == (x) proto_device *x; */

proto_device *linkvtd ( .. ):
    creates a VTD, given class and directory;
    returns NULL if unsuccessful;
```

```

unlkvtd (*vtd):
    destroys the VTD, reclaiming the memory;

initvtd (*vtd):
    initialises the INTTABLE VTD;

exitvtd (*vtd):
    terminates the INTTABLE VTD;

char getcvtd (*vtd):
    gets character from INFLOW VTD;

putcvtd (*vtd, c):
    puts character to the output VTD.

```

2.3. The Prototext Virtual Text Terminals:

The discussion of the Prototext language in the succeeding sections will clarify that the intrinsic VTT repertoire defined below does not limit the choice of VTT to an application. The VTT interface is designed to meet requirement 1.2.5, by tracking syllable boundaries for use with cursor positioning in terms of syllables.

Ascii lead-in & hex value		semantics
BS	08	step back by one syllable
HT	09	clear rest of row
LF	0a	step down
VT	0b	= CR + LF
FF	0c	clear page
CR	0d	set cursor at left margin
DC1 r c	11 <r><c>	set cursor at row, col
DC2 m	12 <m>	move cursor in direction : u d l r : one step, U D L R : to margin
DC3 s	13 <s>	set scroll options, bits :
	0:	reverse LF at left margin,
	1:	reverse CR at left margin,
	2:	LF at right margin,
	3:	CR at right margin,
	4:	roll page down at top end,
	5:	feedback at top margin,
	6:	roll page up at bottom end,
	7:	feedback at bottom margin.
DC4	14	feedback: direction is on top of stack
ESC	1b	divert further text back to the application

In the above, the parameters are all byte-wide characters. The cursor movement parameters are for the directions up, down, left and right respectively. The feedback is enabled by the scroll options and triggered when the top or the bottom margin is hit by an implicit scroll. The device-specific action for the scrolling may then be taken in Prototext. The diversion of text, which I call "reformation channel", may be started by the application, which must terminate it. It is also always temporarily invoked at the top and the bottom margins.

2.4. Prototext architecture:

The Prototext *interface* interprets definitions of VTD, which are directories. A VTD program consists of the init, the main and the exit programs. Init and exit programs are respectively invoked by the `initvtd` and `exitvtd` calls, and must be omitted for `unINITABLE` VTDs. Main is used by `getcvt` and `putcvt` calls. Note that output from a Prototext program is to the device, except in `getcvt`, in which the Prototext "write" serves to fetch the character.

A Prototext program is like a channel, with input text percolating from the top downwards. The program cannot directly select its data source nor terminate the channel. Prototext is specifically designed to reflect the symmetry of the device control problem; each provision selects a facet. Anything else would have been a proverbial jump from the frying pan into C-level interpretation, making things slow and not really solving anything at all.

No new objects may be defined. Certain primitive operations may be redefined in the init program. Reentrance is not allowed. Dynamic programming by self-modification or assimilation of strings is not possible. Objects directly accessible are the current character and the current state, both single bytes. Other objects are the 256 byte ring buffer stack, the syllable/graphic-row buffer, redefinitions of the primitives for graphic VTD, and the "syllable page" holding syllable coordinates in the VTT.

2.5. Virtual Text Channels are INFLOW VTD:

Prototext VTD may be used to make portions of the application programmable, by constructing internal channels, classified as `unINITABLE`. A good way is to define the VTC as `INFLOW`, and defining a sufficient input buffer as illustrated below in C:

```
#include < proto.h >                /* Prototext defines */

proto_device  *vtd;                 /* vtd pointer */
char          ch;                   /* output of vtd to us */
char myrawin ( ) {                  /* input to vtd */
    }
/* in main ( ) : */
    vtc = linkvtd (INFLOW,          /* class */
                  "/u/guru/my_vtd", /* directory */
                  4,                /* buff[16] (log size) */
                  myerrfun,         /* reporter */
                  0, 0, 0, 0,       /* misc stuff */
                  myrawin,         /* use this */
                  0);              /* no raw out */
/* elsewhere : */
    while (ch = getcvt (vtd)) /* programmable */
        { /* fixed processing rest of the way */ }
```

`INFLOW` VTD may be used for "soft keyboards". In practice, the keyboard is part of the terminal, which often supports its own set of soft keyboards. These could be invoked in the init program, as shown below for Fortune terminals (Protile, my Prototext compiler, depends on the C preprocessor "cpp" for all symbolic work):

```
#include < stdef.h >
#include < forkeys.h >
BEGIN { send ALTKEYB1 } END    ; init=> device
```

`Stdef.h`, in `/u/text/include`, contains:

```
#define BEGIN {
```

```
#define END )
#define WRCHR(x) { set x } { write }
#define TERPRI WRCHR(c:10)
```

Forkeys.h contains, among other things:

```
#define ALTKEYB1 "~\kM" /* FS k M - Greek */
```

The following sample program usable with the above provides a Virtual Keyboard with Greek to the application on a Fortune terminal:

```
#include < stddef.h >
#include < ascii.h >
#include < string.h >                                /* macros for Prototext */
/* SEQLDR : sequence leader char;
   SEQCHR : in-sequence char;
   SEQEND : terminator char;
   ENDSEQS : end of all sequences. */

BEGIN { send ALTKEYB1 } END ; init : set to Greek

BEGIN                                ; main part
  SEQLDR (SOH, c:1) ; Fortune function keys
  /* result state is 1 */
  SEQEND (LF, c:2, WRCHR (cx82))
  /* in state 2, resets, emitting hex 82 */
  ; other seq ends
  SEQCHR (cx78, c:1, c:2) ; for up-arrow key
  /* in state 1, go to 2 upon hex 78 */
  ; other seq chars
  ENDSEQS
END                                ; of main part
BEGIN { send ALTKEYB0 } END ; exit : to English
```

A macro defined in string.h is reproduced below:

```
#define SEQEND (x,s,y) \
{ char x                /* if char x */ \
  { state s            /* if state s */ \
    y                  /* action */ \
    { loop }          /* clears stack */ \
    { setst c:0 }     /* reset */ \
    { done }}}        /* with char */
```

An important control mechanism, the anchor, is illustrated next. It obviates switching primitives like "if" and "case".

```
#define A cx41
#define F cx46
; to reverse any seq of A..F within the text
{ anchor }                                ; drop anchor
{ { range A F                             ; anchor clings to levels
  { push }                                ; stack up
```

```

        { rise }                ; continue at
    }                            ; the anchored level
{ push }                        ; other characters
{ rotup }                      ; report in right seq
{ loopop                       ; while stack non-empty
    { write }                  ; emit
}}                             ; end of nested level.

```

The primitive { loop <prog> } runs with characters flushed from the base of the stack, while { loopop <prog> } does it with popping. The stack is also cleared.

2.6. Graphic VTD:

An important aspect with graphic VTD is the separation of device control and the VTD definition, including the syllable grammar defined using { include } and { occlude }, to fetch the glyph appropriate to the current character into the syllable buffer. Relative offset in the buffer and rectangular zones of the glyph may be specified. The syllable is terminated with either { step } or { flush }. The latter clears the buffer. The syllable buffer serves as a graphic row buffer, with { step }, for multi-pass printing, as necessary with any decent font on a matrix printer. A graphic primitive may be redefined within the init program. A typical redef for PLOTTING is:

```
{ define flush { loopsyll { plot } } }.
```

The fragment below illustrates the downloading of a font to a Programmable Character Device:

```

{ set cx0 }                    ; no font for cx0 .. cx1f
{ push }
{ loopop                       ; demonstrate loop
    { occlude }                ; overwrite with new glyph
    { push }                   ; preserve during flush
    { flush }                  ; may be redefined
    { pop }                    ; retrieve char
    { addc c:1 }                ; add decimal unity
    { range cx0 cx5f           ; Ascii graphic range only
        { push }               ; to continue loop
    }}

```

For a downloading situation, the redefinition of flush would be for formatting appropriate to the device, in place of the { plot }.

Important with graphic VTD is { scroll }, which acts upon the current character value. The explicit action is mainly for use with VTT, and with non-VTT graphic VTD only as redefined. Within the redefinition, { scroll } invokes intrinsic action, and is the only primitive legal within its own redef. Outside, a scroll always seeks the redef first. Scroll is implicitly invoked in PLOTTING devices at page margins.

Internal scroll action for DISPLAY VTDs is defined by the VTT. Action in PLOTTING devices is defined for LF, CR and FF (with wrap around). Since scrolling is parametrised by the current char, which is accessible in Prototext, the application clearly has unlimited freedom of choice in its VTT. Application VTT and device capabilities are matched through the intrinsic VTT and the redef of { scroll }.

The separation of the syllable grammar from device control is illustrated below:

```

; main : syllable grammar                : in script directory
; muck containing the following code :

```


The "Session Tty" Manager

S.M. Bellovin

ulysses!smb

AT&T Bell Laboratories

Murray Hill, NJ 07974

ABSTRACT

In many UNIX® systems, it is possible for a program to retain access to the login terminal after the user has logged out. This poses obvious security risks and can also confuse the modem control signals. We solve this for System V by adding a layer of indirection known as the *session tty* driver. At login time, a session device is linked to the physical terminal. User programs have access to the session device only, and may not open the physical line. Upon logout or carrier drop, the link is severed. New login sessions are given new session devices, and are thus insulated from persistent processes. Use of session devices is controlled by a new system process known as the *session manager*; by means of suitable plumbing primitives, a "reconnect after line drop" facility can easily be implemented.

1. INTRODUCTION

*"Any software problem can be solved
by adding another layer of indirection."*

When a user logs on to a UNIX® system, a shell is fired up with file descriptors 0, 1, and 2 connected to the physical tty device used. All commands executed by that user are descendants of the shell, and normally use the same tty device for input and output. These associations — so fundamental to the design of the system — cause trouble under certain circumstances. These problems may be solved by adding a level of indirection called the "session manager".

Before we discuss the solution, it is, of course, helpful to know what problems it claims to cure. There are several, mostly having to do with ending conditions:

1. When the physical device receives an external close indication — i.e., when carrier has dropped, the user's terminal has been turned off or disconnected, or a network close request is received — this fact must be propagated to all processes using that terminal. It is not sufficient to send `SIGHUP`. Some processes may have the signal ignored; others may not have the device as their controlling tty, and hence will not receive the signal at all. In either case, the result is the same: the next user of the line can receive some strange and wondrous garbage.
2. When the login shell — the process group leader — exits, the "session" (whatever that is) should end. Access rights to the terminal by any child processes should be revoked, regardless of whether the physical connection has been broken. If we are dealing with a network connection, such a disassociation may be needed to complete the protocol `close` processing. The system currently attempts to deal with this by sending `SIGHUP` when a process group leader ends; as noted, this is often insufficient.
3. When the host signals external equipment (i.e., modems) that a terminal session has ended, typically by dropping DTR, it must then immediately spawn a new `getty` process, and re-enable DTR. Failure to do either causes trouble: some other process may try to read from

the line instead of `getty`¹, or the line may appear to be dead, thus blocking a telephone “hunt group”.

Historically, these requirements have been a very fruitful source of bugs. Most versions of the UNIX system either suffer from such bugs, contain large amounts of code to try to avoid them (i.e., `forcfclose()` in the 9th Edition and 4.2BSD kernels), or both. None of these solutions is particularly satisfactory.

The session tty mechanism avoids all of these problems by decoupling the physical tty device from the tty device visible to the user. A logical connection is established at login time; it is severed when either the physical device notices that the connection has dropped, or when the login shell exits.

In addition to solving the original problem, there are several other benefits to session ttys:

1. Permitting a user to reconnect to a session after a physical hangup is easy.
2. Remote login sessions can be assigned “tty” names for `/etc/utmp` entries; thus, they will show up via `who`.
3. The `/etc/utmp` entry for the login session will remain around while any child processes persist. This helps system administrators track down hidden resource consumers.
4. Resources allocated to a session — say, a tape drive assigned to a user for the duration of several commands — should be revoked when the session ends. In current systems, it is very hard to determine when this is true.
5. Certain stream modules whose usage must be beyond the user’s control (i.e., encryption or audit modules) can be protected from tampering.
6. Non-login sessions — circuit-based network connections, *cron* jobs, etc. — can have `utmp` entries as well.
7. Depending on certain design questions that are not yet clear, session ttys may provide a clean solution to the problem of which window is logged in when using *layers*.
8. Again, depending on certain as-yet unresolved questions, use of tty lines for two-way traffic may become significantly cleaner and easier.

The implementation discussed here is for System V Release 3, as it heavily relies on *streams*^[1] and on several other unique features, such as multiplexor devices and clone devices.² Some features not in the standard system, such as a streams-based terminal line discipline, are also used. It is possible to implement the session manager on other versions of the UNIX system, but it is somewhat more difficult to do so.

2. HOW IT’S DONE

There are three key components to the new design: the *session manager*, a multiplexed device driver known as *sesspty*, and one or more *line managers*. Depending on the exact hardware and software configuration, the line manager may be a separate process, part of an existing “listener” process, or part of the session manager itself.

2.1 The Current Structure

Before we discuss the new implementation, let us review the current login management structure. When the system makes a transition to multi-user state, `/etc/init` creates a child process for

1. The security implications of this are amusing; it allows a process to imitate `getty` and `login`, thus capturing users’ passwords.

2. See the glossary at the end.

each tty line, as defined in `/etc/inittab`; each child process in turn executes `/etc/getty`, which opens the actual device. When the open succeeds, `getty` sets up the hardware environment (principally the line speed), collects the user's login id, and passes control to `/bin/login`. As noted earlier, the physical device opened by `getty` is passed directly to `login`. `Login` validates the login id (i.e., verifies a password), creates an entry in `/etc/utmp`³, sets up the user's environment, and passes control to the user's shell. The only `fork()` operation in the entire process is the one performed by `init`; the user's shell is thus a direct descendant of `init`, with a process id known to `init`. We refer to that process as a *session process*. When the shell terminates — that is, when the user logs off — `init` is notified of the process id; it cleans up the `utmp` entry and spawns another `getty` process for that line.

The following table summarizes the process:

TABLE 1. Division of Responsibility

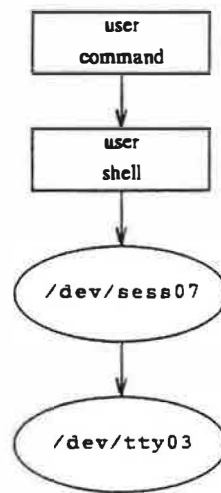
<i>Program</i>	<i>Function</i>	<i>External triggers</i>
<code>/etc/init</code>	Fork and invoke <code>getty</code> Clean up <code>/etc/utmp</code> entry	Previous session process died
<code>/etc/getty</code>	Hardware environment setup Collect login id	Line open succeeds
<code>/bin/login</code>	Validate login id Software environment setup Create <code>/etc/utmp</code> entry	
<code>/bin/sh</code>	(user session) exit	User logs off or line drops

Network tty connections do not differ in any fundamental way. `/etc/init` is no longer in the picture; the login shell will be child of some network listener process. It in turn is responsible for cleaning up any `utmp` entries. `Getty` may or may not be used.

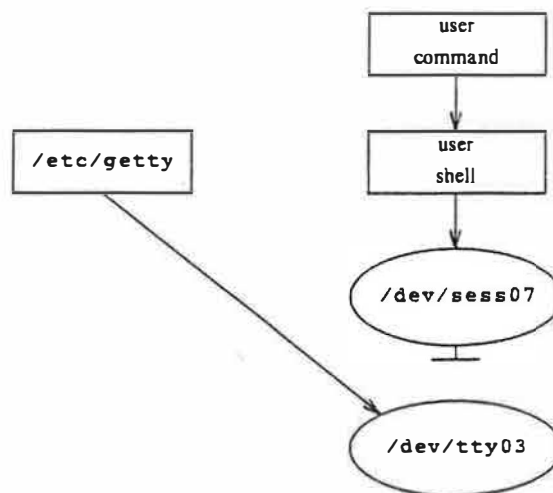
2.2 A Brave New World

The primary conceptual change in the new system is that the physical device is no longer passed directly to `login`. Rather, a software pseudo-device — `sesstty`, as you probably guessed — is opened by `getty`. By dint of the `clone` device driver, a separate `sesstty` device is used for each session. Next, the physical line is spliced to the `sesstty` device by the `I_LINK ioctl()` call. For the duration of this session, all references to the line, including its `/etc/utmp` entry, are via the `sesstty` device instead. The following diagram summarizes the situation:

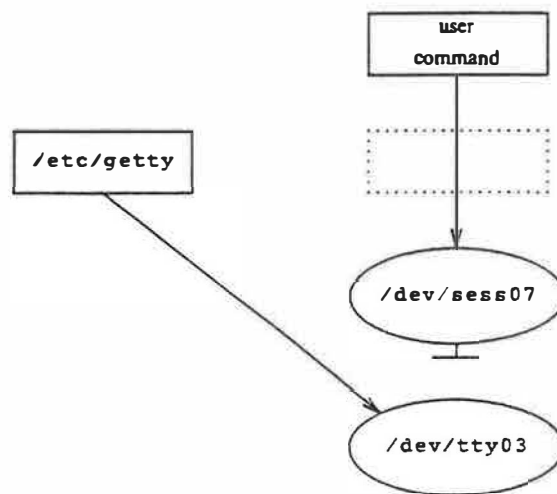
3. Strictly speaking, the `/etc/utmp` entry is created by `init`. This is primarily a bookkeeping entry, however; the significant entry — i.e., the one showing that someone has logged in — is created by `login`.



When the physical line drops, the connection between the session device and the physical device is broken. As shown below, a new invocation of `getty` can attempt to reopen the line, without affecting the old session device still being used.



The connection is also severed if the shell exits while any of its children are still running:



Although the new scheme appears to be a minor enhancement to the current system, there are actually several complicating factors. For one thing, we have added some new external events. A new `getty` process must be spawned when the link is severed; this may or may not be coincident with the demise of the session process. Second, session cleanup, notably clearing the `/etc/utmp` entry, occurs when the `sesstty` device is closed, which in turn will not normally happen until all child processes terminate. Both are currently handled by `/etc/init`; however, it seems unwise to further complicate `init` further by adding new functionality to it.

We solve these problems by divesting two of `init`'s roles to some new programs, the *session manager* and the *line manager*. `init` will fire up a single session manager, and one or more line managers, in accordance with the current system state. The line manager is similar to the current `/etc/getty`. Its primary task is to open new tty lines; additionally, it handles hardware environment setup and collects and validates the login id. After that, however, it does not invoke `login`; rather, it opens a stream to the session manager and issues `I_SENDFD ioctl()` calls to pass the physical line to the session manager. It never invokes an actual shell. A typical implementation would fork a child for each line or network connection; the child would exit when the line had been handed to the session manager.

In general, there may be an arbitrary number of line managers. One might handle all ordinary tty lines, a second might handle remote login requests as part of a "listener" process, etc. If the line manager is some sort of listener process, no special action need be taken to re-enable the line after it has dropped. The situation is somewhat more complex if we are dealing with ordinary tty lines, where each port must be opened separately. In such cases, an explicit message must be sent to the line manager by the session manager.

The session manager is the parent of all user shells, thereby relieving `/etc/init` of the responsibility. At initialization time, it opens the *control channel* of `sesstty`. When the session manager receives a file descriptor from a line manager, it links it to a session device, creates the initial `utmp` entry, and forks to a replacement for `/bin/login`, known as `/bin/loginenv`. This program — which is not privileged — is responsible for setting up the user environment before passing control to the shell. When the shell terminates, the link between the physical device and the session device is severed (which implies a `close()` of the physical device will take place), and the line manager is advised by the session manager to re-enable the line.

The session manager also receives control messages from `sess tty` telling of physical line drops. Again, the connection to the session device is severed and the line manager appraised of the situation. In neither case, though, is the `utmp` entry cleared; that happens only when the session device is closed. Such status changes are passed to the session manager via the control channel.

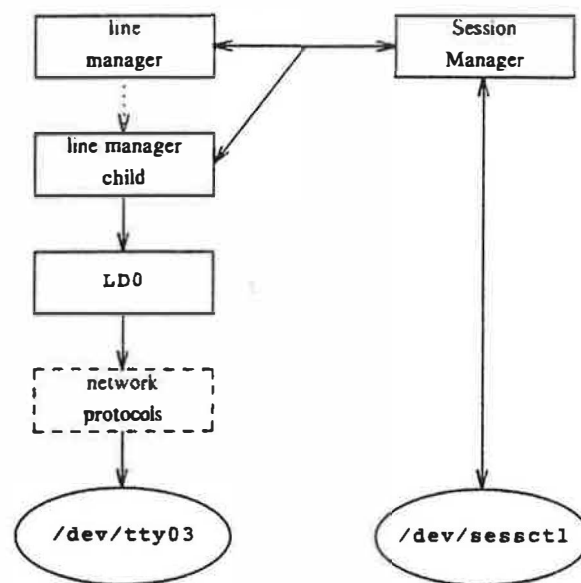
`Sess tty`, although one driver, is divided into two logical sections. The first section handles the control channel; very little data is actually sent on this channel. It is used to pass status change messages to the session manager, and as the device for `I_LINK` and `I_UNLINK` calls. The second section is the mostly-transparent user device handler. Most data and control messages are passed through, whether upstream or downstream. Hangup messages from the real device are diverted to the control channel, though, as are close messages from the stream head.

2.3 The Gory Details

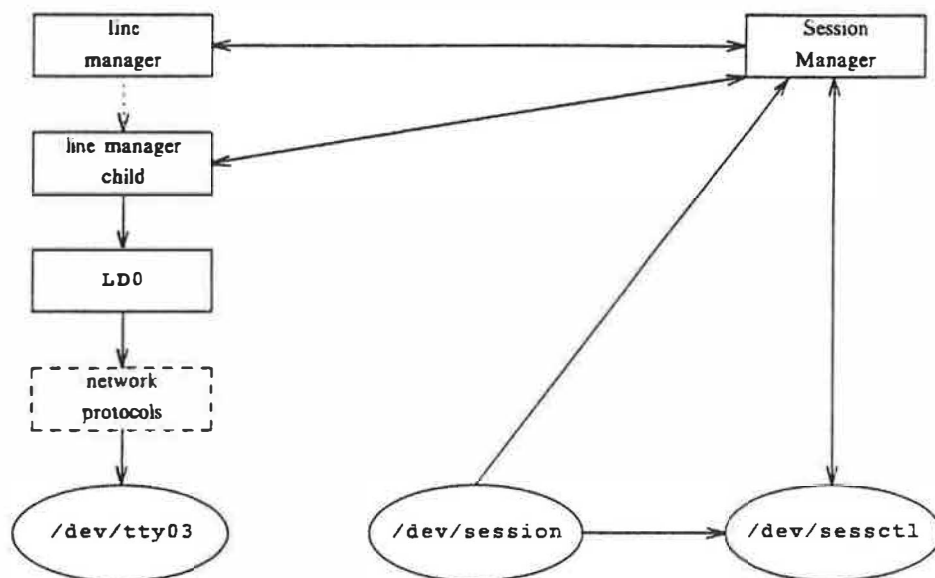
Let us now re-examine the entire structure, this time paying attention to all of the small details. We will do this by examining the boot-time initialization procedures, and then following the progress of a login session from beginning to end, with occasional diversions to discuss what a component is doing while awaiting an event.

At boot time, the session manager opens the session driver control channel and creates a *named stream pipe* for communication with the line managers. Each line manager creates a new stream pipe, opens the named pipe to the session manager, and passes one end of the stream pipe to the session manager. Thus, each line manager has a unique channel to the session manager.

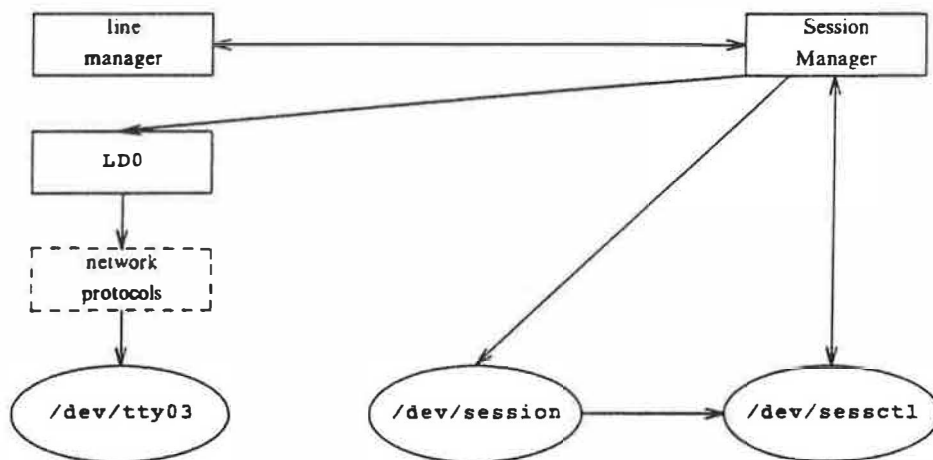
A session starts when the line manager — we will assume for the moment some form of “super-getty” — detects that an `open()` attempt on some line has succeeded. The line manager will then `fork()` a per-line child process, to handle the negotiations with the user and the session manager. The child process will collect the login id, much as `getty` does now. It will then *push* LD0 (the standard tty line discipline), issue the appropriate `TCSETA` call to configure the line, and invoke an authentication mechanism.



Following that, the line manager child creates a new stream pipe and passes it to the session manager, thereby creating a unique channel for each session. This channel is used to pass login information from the line manager child to the session manager, and to send it the file descriptor for the physical line via `I_SENDFD`. The session manager opens the clone device `/dev/session` to create the control terminal for the new session.



At this point, the per-line process may exit.



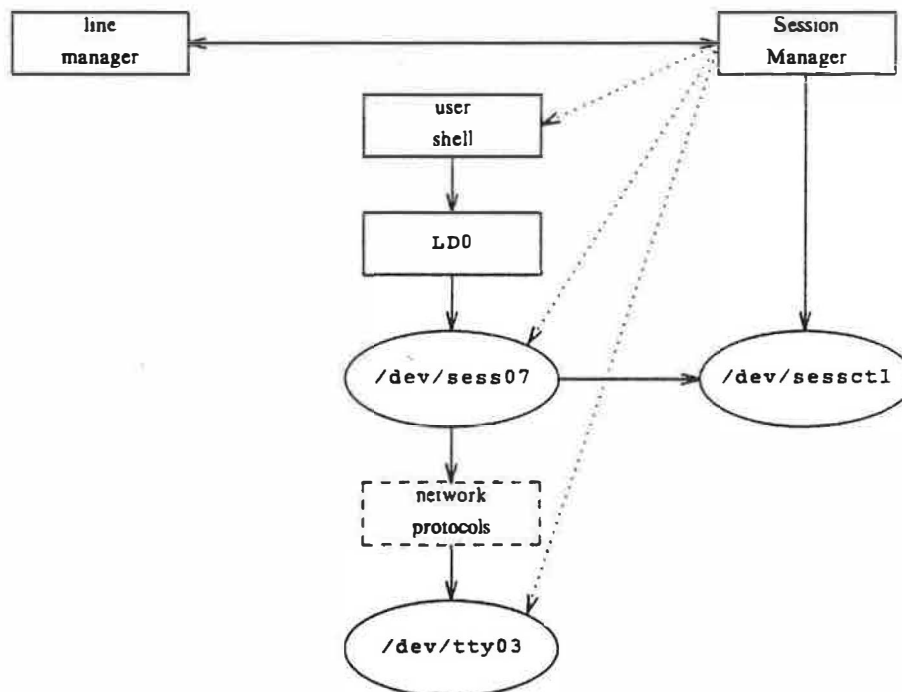
The scenario is similar if we are using a listener-based line manager. The primary difference is that any network-related line disciplines must be pushed onto the stream before LD0. As we shall see, the exact set of disciplines used must be carefully selected, as they will move out of reach of the user before the login process is complete.

Most of the time, the session manager is in a loop waiting for messages on the control channel. Eventually, it will receive a validated login id and a physical device on this channel. At this point, a `utmp` entry is created and a child process is created. This child process must shed any current control tty (i.e., the control tty of the line manager) and open the session device to acquire a new control tty. It then performs a `TCGETA` to note the line environment, pops LD0, issues an `I_LINK` to splice the physical line to the session device, and pushes LD0 onto the session device. This juggling of LD0 is necessary because line disciplines below the splice point — say, a network protocol module — may not be popped once the splice has taken place. But many programs will want to remove LD0 on their own; it is not reasonable to isolate it behind this brick wall. (The wall is actually useful for some applications. The line manager may wish to push a cryptographic module; for security reasons, it may be desirable to prevent the user from ever removing it.)

Finally, the child process `exec`s the user's shell.

Once the fork has taken place, the session manager should close the physical device passed to it by the line manager. For each session, it must record the session device, the real device, the “magic cookie” passed back by the `I_LINK` call, and the process id of the child, which will ultimately become the user's shell.

After all of the juggling has taken place, the configuration looks like this. The dotted arrows indicate information; the solid arrows indicate an open file descriptor or some other active data channel.



As noted, the `sesstty` driver passes most data and control messages through without modification or hindrance. An exception is made for one event: an `M_HANGUP` indication coming from the real device. `M_HANGUP` is an actual stream message, and may be diverted if we choose. If left to propagate upward, it will cause `SIGHUP` to be generated; as we have seen, though, this may not accomplish everything we want. Accordingly, the message is diverted, and a status message is sent instead to the session manager. The session manager in turn issues `I_UNLINK`, to sever the connection to the real device; if necessary, it also notifies the line manager to re-enable the device. Finally, the session manager must issue a command (via the control channel) to cause a real `M_HANGUP` message to be sent to the session device. (We shall see later why we do not simply make a copy of the `M_HANGUP` to send to the session manager, rather than diverting it.)

Setting up a session is straight-forward. Tearing one down, however, depends on three independent events: termination of the session process (which we will refer to as `PEND`), a close of the session device (`SCLOSE`), and a `M_HANGUP` of the real device. These may occur in any order. If we judiciously use `sighold()` to block `SIGCLD`, all of these events will appear to be synchronous to the session manager, thus avoiding potential race conditions.

The cleanup rules can be stated fairly simply.

1. When the first termination event comes in for any session, sever the connection between the session device and the real device, and tell the line manager to re-enable the line.
2. If the first event is `M_HANGUP`, send an `M_HANGUP` message up the session device to the user processes.
3. When both `PEND` and `SCLOSE` have occurred, the session is over; clean up the `utmp` entry.
4. Whenever `M_HANGUP` arrives, send a re-enable message. Hardware hangup signals are external to the machine, and hence asynchronous; depending on internal timing, one may arrive while a re-enable is in transit, and hence its effect may be lost.

We must also deal with the problem of commands that (a) close all of their file descriptors, and (b) persist after the shell itself has exited. This would cause the session device to be closed, which in turn would cause the session entry in `/etc/utmp` to be deleted. Such processes would then become invisible in some sense. We deal with this by changing the implementation of `/dev/tty` slightly. Rather than storing a pointer to the `tty` structure in the `"u."` area, we instead store a pointer to the file table entry for the session device. That is, when a process acquires a session device as its controlling terminal, the kernel would in effect perform a `dup()` operation on the relevant file descriptor, storing the result in `u.u_ttyd`. Thus, the session device could not close until all processes in that session have terminated. (The close notification may be delayed even further, if other process outside the session are using the device. The `write` command is one case in point.)

There is one anomalous case of line re-enabling that must be dealt with. If the child of the line manager does not validate the login id successfully, it will never hand off the connection to the session manager. Consequently, no ordinary re-enable message will ever be sent. There are a number of ways around this problem, such as having the line managers look for special exit codes from their children, or having the children always send re-enable requests to the parent via the session manager.

Let us revisit Table 1 and examine the new division of functionality:

TABLE 2. New Division of Responsibility

<i>Program</i>	<i>Function</i>	<i>External triggers</i>
/etc/init	Fork and invoke session manager and line managers	Previous process died
Line manager	Contact session manager Open set of lines Fork per-line child	Line open succeeds
Line manager child	Hardware environment setup Collect and validate login id Hand off line to session manager	
Session manager	Create session device Create utmp entry Fork/exec loginenv Disconnect session Clean up utmp	Message from line manager Message from line manager child Line hangup or child process exit Session ends
/etc/loginenv	Software environment setup Exec user shell	
/bin/sh	(user session) exit receive SIGHUP	User logs off session manager message (after session disconnect)

3. BELLS, GONGS, AND WHISTLES

In addition to fixing some bugs, using the session manager makes it easy to add some new functionality to the system. The most interesting new feature is the ability to have a session survive the disconnection of the physical terminal. Instead of sending `M_HANGUP` to a session when the physical line drops, nothing would be done. Read requests on the session device would eventually block due to lack of input data; write requests would block because of flow control on the stream. Alternatively, a `STOP` signal could be sent to the session's process group instead; that way, a user who disconnected because of stubborn malfunctioning processes would not have to worry about them eating the CPU alive until they were killed manually.

To resume a suspended session, a user would log in and issue a `reconnect` command. This command would ask the session manager to sever the link between the real device and the new session, and then splice the real device to the `sesstty` from the old session. The new session would probably be terminated, though as an option it, too, could be suspended. This would allow a user to switch back and forth between multiple sessions, possibly even on different machines.⁴

It isn't clear how long a disconnected session should be allowed to persist. Permitting such session to live indefinitely would tend to choke a system on the bodies of undead⁵ processes. Presumably, a timeout interval would be set by the system administrator; users who wished to suspend a session for a longer time could override the limit via an `ioctl()` call.

Instead of having the output of a disconnected session pile up, we may wish to discard it. The obvious model is a "streams `/dev/null`"; unfortunately, that's not as simple as it sounds. It's certainly easy enough to implement a pseudo-driver that discards any data flowing into it; however, there is no way for a driver to pass a "hard EOF" back to the stream head. If such an

4. Any resemblance between this concept and the `sxt` driver is *not* coincidental; we discuss the similarities below.

5. I can't call them zombies...

enhancement were made, the session device could be spliced to a clone of the null stream. Alternatively, some large number of 0-length records could be sent upstream.

Earlier, we alluded to the resource allocation problem. Briefly, a user may wish to acquire some resource for a span longer than a single command. A tape drive is a good example; the user may need to create a `cpio` image on the tape, verify it, recreate it when the verification shows that the file is incorrect, etc. In a closed-shop environment, assignment of the tape drive to that user may be handled by a tape drive manager daemon. Apart from allocating the drive, this daemon would also request that the operator mount the user's tape. The problem is when to release the drive for allocation to other users. While a time limit is one solution (and probably part of any solution), noting when a session has ended is another. Today, detecting logout is insufficient; the process using the drive may be running in the background. Sessions, however, persist until the last process is gone.

Session-based ttys are also useful to B2-level secure UNIX systems^[2]. Such systems require the existence of a *trusted path* — a mechanism invoked by the user that will reliably connect that user to a trusted program. The trusted path must be used at login time; that way, the user is guaranteed that the `Password:` prompt is not coming from a Trojan horse program. Obviously, a requirement for the trusted path is that all other processes talking to the terminal must be killed or have their access rights revoked. Sheerly in terms of implementation problems, this is a complex task. The list of open files for a process is in the process's pageable “u.” area; thus, it is not easily accessible to the kernel at interrupt time. Session ttys inherently solve the problem. When carrier drops, an `M_HANGUP` message is generated. Upon receipt of the message, the session manager will *always* disconnect the physical device from the session device, and *always* ask the line manager — a trusted process — to re-enable the line. The only change needed to the session manager is that the line may not be re-enabled until `M_HANGUP` is received; presenting a new `login:` prompt upon session head exit violates the DoD requirement that the user must initiate access to the trusted path.

Session heads need not be bound to tty devices; rather, they can be linked to any underlying stream. This gives us the ability to bring other services under the same umbrella. For example, RFS mount operations could be listed in `/etc/utmp`; that would permit easy monitoring of start times, exit status, etc. Other network sessions — file transfer operations, remote execution requests, etc. — could be listed as well. Some care should be exercised in deciding what types of sessions should be listed, of course; things like directory queries or time server queries are likely to be too transient to be worth the overhead.

One essential difference between the current `/etc/init` and a line manager is that the latter knows when an active session is in progress. All `init` can say is that something is running; it cannot tell whether or not the line is actually open. But this knowledge is extremely important to programs that wish to use the line for outgoing calls. Current solutions are not particularly clean. `Uucico`, for example, requires a special version of `getty` known as `uugetty` for shared lines. A better solution would be for it to ask the line manager for a line — any available line — rather than relying on heuristics to determine what is happening. Ideally, in fact, such negotiations would be handled by a connection agent, rather than by each individual client program.

4. ERROR RECOVERY

A serious disadvantage to the session manager is that it creates new critical processes. If the session manager crashes, or if a line manager gets confused, it would be impossible to log in to the system. It is thus necessary to plan for such occurrences.

It is comparatively easy to recover from a line manager crash. A tty line manager needs one essential piece of information: which lines are currently in use, and hence which other lines should be enabled for login. We accomplish this at line manager startup by having it “log in” to the session manager. As part of this dialog, it indicates whether or not it cares about lines currently in use; if it does care, the session manager will send it a list. Ports assigned to that line manager are enabled if not currently in use.

Network-based line managers are much simpler. They rarely care which ports are in use; that is generally the concern of the transport layer. Thus, their login sequence does not request notification.

In either case, it is possible to kill and restart the line manager without loss of state. This suggests that for redundancy, at least two line managers should run on each system. Typically, one would handle the console only, while a second would handle all other tty lines. Additional line managers would be used for each network type.

Recovery from a session manager crash is more difficult. When the session manager crashes, the `sessctl` device is closed, thereby automatically unlinking all of the session devices from the physical lines. Users are disconnected, but their processes can remain suspended; when the session manager is restarted, reconnection is possible. Unfortunately, if the session manager is the sole means of access to a system, it isn't possible to log in and restart it. Some crashes can be handled by having a daemon periodically query the session manager, and restart it if necessary; obviously, though, there are many failure modes immune to this sort of automated detection. Perhaps the best solution is to have a standing network server that will, on request, kill and restart the session manager.⁶

5. SESSION DEVICES WITHOUT SESSION MANAGERS

In some environments, a simplified version of this mechanism can be used. We can eliminate the session manager and the line managers, and simply change `getty` to talk to the session driver.

Assume that all access to the system is via a network, or via a port selector that will connect a user to any free port that has DTR enabled. Assume further that there are sufficiently many ports on the system that a short-term loss of some is not a serious problem, but that revoking access to the terminal is important. In that case, when `getty` answers a line, it would open the clone driver `/dev/session` and splice the physical line below it. All further login proceedings would take place on the session device.

We now have two end conditions: the login shell exiting, and carrier dropping on the physical line. When the former happens, a new `getty` will be spawned by `init`. If there are no left-over processes, the session device will be closed, which in turn will cause the physical line to be unlinked. The `getty` can then proceed normally. If there are left-over processes, `getty` would open the old session device and manually unlink the physical line; life could then proceed as before.

Carrier drops are more problematic. If the login shell does not exit, no new `getty` is spawned; hence no process issues the `I_UNLINK` call. However, the `M_HANGUP` message is detectable by the session driver. It cannot unlink the line — for assorted complex reasons, that cannot be done at interrupt time — but it can block output to the physical device from the session device. Additionally, the physical device driver can be modified to drop DTR if carrier drops; thus, the line will not be available for reuse until it is reopened by some future `getty`.

This simplified version is very susceptible to denial of service attacks. An enemy could log in and then drop the line, thereby tying up a session device. Some provision must be made to detect too many session devices being assigned to one user.

Despite appearances, it may be possible to use this version with a modem pool. Many modems can be configured to busy out the line if DTR is low; thus, a disabled line will be skipped by the phone network.

6. A better solution could be contrived if someone would bring back console bit switches and lights....

6. IMPACT ON OTHER SYSTEM COMPONENTS

The session driver requires remarkably few changes to other system components. Obviously, the entire implementation depends on the tty subsystem being converted to use streams instead of `clists`. Most of the changes are to `getty` and `login`. At a minimum, the version of `login` invoked by `getty` must be changed to communicate with the session manager after collecting the login name. More likely, we could remove the password-checking code from `login`, and let `getty` — a line manager — invoke an authentication routine. It is important to perform authentication before discarding knowledge of the physical device.

`Getty` also needs to be integrated with a line manager. That can be done in either of two ways. First, we could write a separate line manager program; it would fork and invoke per-line `getty` processes, much as `init` does now. Alternatively, we could write a “super-`getty`”, a program that would have `open()` requests outstanding on multiple lines at once, and would only `fork()` when the `open()` succeeded. This latter approach reduces clutter in the system process table, at the cost of greater complexity and lesser robustness. Super-`getty` is the only solution if we wish to implement two-way lines.

If `login` loses its authentication code, and hence becomes unprivileged, the code in `sh` that `exec()`s `/bin/login` must be changed. Conceptually, what must be done is to disconnect from the current session, while asking the line manager to re-enable the line without dropping it first. It is quite conceivable that some line managers will not be willing or able to do this. More thought needs to be given to this question.

Existing listener processes will need to be converted to talk to the session manager. Currently, there are only two major ones for login sessions, the DKHOST listener for the Datakit® VCS and `/usr/net/listen` when used with STARLAN.

The `/dev/tty` change described earlier is actually independent of the other changes described here. In fact, it could be done with the current system, though there would be added complexity when handling end conditions such as an orphaned background process. Still, those problems are not insurmountable. Apart from changes to the `/dev/tty` driver itself, the `fork()` code would have to be changed to perform the `dup()` operation, and `exit()` would have to perform an additional `close()`.

The current accounting routine records the device number of the controlling tty. Under sessions, that value would be the device number of the session head, which has no physical reality. If the physical device number is desired there, some mechanism to announce it must be provided. On the other hand, it may not pay to change that; network connections have no true physical device number available, and the session device number serves to group together commands from a given session.

It would be nice to add some new fields to the `utmp` structure, to record a link to the physical device or source host; compatibility considerations may preclude this, however. If `utmp` cannot be changed, a parallel file containing this information must be set up. A few more type codes for `utmp` are certainly needed to indicate disconnected sessions and the like.

Use of the session manager will exacerbate an existing incompatibility. There are a few programs, including some versions of the Korn shell, that use `getppid()` to determine if they are direct children of `init`. Under this new scheme, almost nothing will be. Of course, the same is true now for children of listener processes.

7. FUTURE DIRECTIONS

As noted, the ability to bounce back and forth between different sessions is very similar to functionality provided by the `sxt` driver. Indeed, by rewriting `sh1` we could eliminate the driver entirely. The new `sh1` command would act as a line manager, and set up a new session for each shell layer created. There is some problem with implementing `noloblk` in a disconnected session; one solution is to have `sh1` splice a stream pipe to all of its session heads, and let it perform the multiplexing at user level. A better idea might be to install real job control, and discard `sh1` entirely.

A similar technique could be used to replace the `xt` driver; `layers` could do the multiplexing and handle the line protocol to the DMD itself, much as `mux` in 9th Edition and `mpx` in 4.3BSD do. This would shrink and simplify the kernel, and allow easier modifications to the protocol handler.

There is certainly some efficiency loss in having this multiplexing done at user level. On the other hand, it has always been the UNIX system philosophy that the kernel should provide the basic primitives and plumbing, while letting application programs provide the richness and complexity. This is perhaps best-illustrated by the duplication of code represented by having both `xt` and `sxt` present in the same system.

If efficiency is a major concern, the streams mechanisms provide several handles to resolve the problem. Perhaps the nicest solution for `layers` is to adopt `mux`'s approach: to move the tty line discipline into the terminal. Thus, only complete lines are passed to user level, thereby avoiding the expense of several packets and context switches per character typed. Alternatively, a "buffer" line discipline could delay characters sent from the terminal until a complete packet was received, again reducing the context-switch overhead. (Obviously, such a line discipline would need to turn itself off if a raw-mode application such as `vi` were running.)

Letting each layer be a session has a curious implication: each such layer would (or at least, could) have a `utmp` entry. This is both good and bad. On the one hand, it provides a clearer picture of who is doing what, and eliminates such oddities as the `relogin` command. On the other hand, if each layer appears to be a true login session, commands that scan `utmp` need to be modified to realize this. It isn't pleasant to receive six separate copies of a `wall` message, for example. (It turns out that `wall` is a particularly nasty case. Programs that are split between a DMD and the host, such as `jim` and `cip`, can get very confused if they receive data that isn't part of their protocol. Under the current scheme on System V, that is only a problem if one is sufficiently incautious as to invoke such a program in the designated login window. One can bypass this specific problem by modifying `wall` to ignore lines that are in raw mode; this does not solve the general problem, however.) On balance, we tend to like having each layer appear in `utmp`, but it is certainly a debatable point.

8. IMPLEMENTATION QUIRKS

The *Careful Reader* will have noticed that this design uses several features — stream pipes, named stream pipes, and a stream tty line discipline — that do not appear to be part of System V Release 3. This perception is somewhat correct in practice, and entirely correct in principle.

The stream tty line discipline is a prototype adapted by others from the STARLAN support.

Stream pipes are currently supported in a strange fashion. By opening the clone device `/dev/spx`, a process receives half of a stream pipe. To create a full stream pipe, `/dev/spx` must be opened again, and the two halves spliced together via the `I_FDINSERT ioctl()` call. In the distributed system, only `root` may open `/dev/spx`. The `/dev/spx` driver is not considered to be a public interface, and may not be present in this form in future releases of System V. It does work today.

Creating named stream pipes is even more bizarre. First, one creates a stream pipe. An `fstat()` call is used to determine the major and minor device numbers used for one end of the stream pipe. (Recall that these are actual devices as far as the kernel is concerned.) A `mknod()` call is then used to create a new, named `i-node` corresponding to that pipe end. Other processes may open this filename and receive a file descriptor equivalent to the original pipe end.

All processes that open a named stream pipe receive the same pipe end. That is, data written by them may be intermixed, and no identifying information is transmitted to the far end. Creating unique connections from the line managers to the session manager is a key problem.

We accomplish this by using the named pipe solely to pass file descriptors. That is, each line manager, at startup, creates a new stream pipe and passes one end of it to the session manager via the named pipe. All further communication takes place on this new pipe. A facility similar to

4.2BSD's *UNIX-domain sockets* would simplify matters.

When a session is starting up, each line manager creates yet another stream pipe for the per-line child process to pass to the session manager via the line manager's pipe. This separate connection is created for two reasons. First, there is the aforementioned problem of having multiple simultaneous conversations on a single pipe. Second, when `I_SENDFD` is used to pass file descriptors, these file descriptors are sent immediately to the head of the receiving process's read queue. They are not associated with any data message, nor are they sent synchronously with one. The session manager is thus incapable of telling which physical line goes with which session. 4.2BSD's mechanism for passing access rights to a file along with some text would alleviate this problem.

9. UNSOLVED PROBLEMS

The current design is very heavily oriented towards login-type sessions. That is, it assumes that a shell is the desired end-point of any connection. Although this is certainly true for login sessions, it is not true for some other types of session. For example, a remote execution request would need very different parameters to that shell than an interactive request. An FTAM session would not even use a shell. The current session manager protocol will have to be redesigned or extended so that the line manager (or rather, the program acting as a line manager in this context) can retain ownership of the session device. Some possible new designs would make the session manager disappear entirely as a separate program.

The issue of subsessions, for `layers` in particular, needs more thought. The effect of a disconnection of the physical line must be propagated to all subsessions. Since sessions can only be created by `root`, it may be sufficient to inform the primary session of hangups; on the other hand, privileged programs are not, in general, immune to bugs. If we are using sessions for resource management, we need a way to bind the resource to the real session, rather than any subsession; it is certainly reasonable to use a tape drive in a window other than the one it was allocated from.

The biggest wart in the current design is the nature of the communications channel between the session manager and the line manager. The current scheme requires that a large number of file descriptors be kept open by the session manager; if each invocation of `layers` is in effect a line manager, the session manager could exceed the system limit.

10. SUMMARY

The session manager is a single concept that solves several problems:

1. It introduces a strong concept of a session into the UNIX system.
2. Each session is isolated from all other sessions on the same physical line.
3. The handling of modem control signals can be made reliable, despite ill-behaved user processes.
4. Non-login connections and windows can be treated as sessions, if desired.

11. ACKNOWLEDGEMENTS

I'd like to thank Steve Albert for reading and commenting on an early draft of this document. He, Paul Lustgarten, Nancy Mintz, and Ed Whelan gave me several opportunities to explain the entire concept, thereby helping me clarify the designs; they also helped me focus on the essential parts of it. Additionally, Dennis Ritchie had a number of useful suggestions.

GLOSSARY

- clone driver* A clone driver provides a means to assign a unique minor device to each process using the driver. When a clone device is opened, an idle minor device is selected, and the *i-node* used is modified to point to this minor device rather than the clone device *i-node*.
- I_LINK* An `ioctl()` command used to set up the linkage between a multiplexor driver and a stream.
- I_SENDFD* An `ioctl()` command used to send an open file descriptor across a stream pipe.
- I_UNLINK* An `ioctl()` command used to tear down the linkage between a multiplexor driver and a stream.
- listener* A process that receives and dispatches connection requests arriving via a network.
- multiplexor driver* A multiplexor driver is a pseudo-device driver that permits any open stream to be linked beneath it. It can be used to layer a protocol — say, IP — on top of a physical device. Thus, IP could be implemented as a multiplexor driver, while the Ethernet interface would be a stream to be linked beneath it.
- named stream pipe* A stream pipe with a name in the file system that can be opened by other processes. Akin to a FIFO, but full-duplex.
- session manager* See pp. 1-16.
- stream pipe* Similar to an ordinary pipe, except that it is implemented via streams. Thus, it is full-duplex. A stream pipe may be used to transmit file descriptors via `I_SENDFD` and `I_RECVFD`. It is possible to push modules onto a stream pipe, but that feature is not used by the session manager.
- UNIX-domain socket* A mechanism for intra-machine, inter-process communication on 4.2BSD and 4.3BSD. Addresses in this domain look like file names. Both data and access rights — file descriptors — may be passed across UNIX-domain connections.

REFERENCES

1. Ritchie, D.M. "A Stream Input-Output System". *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, part 2 (October 1984), pp. 1897-1910.
2. DoD Computer Security Center. *DoD Trusted Computer System Evaluation Criteria*, 1983, CSC-STD-001-83.

A low cost bitmapped terminal on the Atari ST

Peter Collinson
Computer Laboratory
University of Kent
Canterbury, UK
pc@ukc.ac.uk

ABSTRACT

The development of workstations with large bitmapped screens and a pointing device has meant a revolution in computing practice. This paper describes a program written for the Atari ST micro which is an attempt to bring some of this revolution to the a wider user base by the use of low cost technology. The program, called *Term*, allows the user to control and maintain a windowing terminal. The environment may be used on BSD UNIX† systems to run several shells supported by a host multiplexer. The terminal uses many features of the Atari ST, including the ability to display differently sized fonts. The paper discusses the various design issues and comments on how well the various system components perform on the low cost hardware.

Introduction

The development of workstations with large bitmapped screens and a pointing device has meant a revolution in computing practice. It is reasonable to speculate that many workstations spend much of their time acting as sophisticated terminals accessing other machines. The selling point is the environment in which the user can work, the ability to manipulate several virtual terminals on one screen while switching data input from the keyboard between several tasks each running on a virtual terminal.

The ability to use virtual terminals is probably 90% of 'why workstations are attractive' when looking from the 'old fashioned' text oriented world based on VDU's. The feature of workstations that makes them unattractive is their price. To replace all the existing text terminals on the Kent campus with a workstation is currently not financially possible. Therefore, it has seemed reasonable to look at alternative methods of supplying the users with multi-windowing environments.

Virtual terminals on text VDU's.

If a user has a text screen and wishes to run some form of windowing then they could use one of the available multiplexers designed to work using *termcap*. The earliest program that was available at Kent was *wm* written by R.Jacob and obtained from Usenet.

There are two main problems with this type of system. First, the systems use too much of the valuable screen area in showing the edges of windows. A lesson taught by visual editors is that screen area is important; even a single extra text line on a screen can enhance the user's work. The shrinkage in screen area by even a single row or column is keenly felt by users and people will swap between the window manager and an editor to avoid the problem. The second problem also relates to lack of available resources. It is hard for the implementors to provide a way of controlling the multiplexer without stealing a key from the overloaded keyboard and perhaps a command line from an already overloaded screen.

At Kent, *wm* has been available to users for some time but it has not become popular. The conclusion is that the program is not widely used because the hardware on which it runs is not suitable for the

† UNIX is a trademark of Bell Laboratories.

task. It is useful to examine what the user would like have to support a windowing environment before discovering what must be sacrificed to reduce the hardware cost.

Suitable hardware

The terminal needs a large high resolution screen. The screen should be supported by bitmap technology so single pixel lines may be used to delineate windows and so that the text is not constrained to appear at any particular position on the screen. A 1024 by 1024 pixel area is desirable.

A keyboard is needed for primary user input. In addition, the user needs to interact directly with the terminal to select windows and carry out other management tasks; a pointing device such as a mouse is the current favourite for this job. A terminal should conventionally run down an RS232 line, but for speed and flexibility some form of fast network connection such as an Ether transceiver should be supplied. The type of processor used to drive the system is irrelevant. A local requirement is the ability to program the system in C, since this is the best assembly language invented to date.

The reality of the hardware specification is that there are no available devices on the market at a price competing with that of a standard VDU. Manufacturers state that the basic price of the large screen is such that it boosts the cost price of the machine out of possible competition. Also, the price of a fast network connection hardware itself equates to that of a normal VDU, so the desire for this may have to be reduced to some form of cheaper external connection.

One might consider building an appropriate device, but it is hard to do this cheaply. To get needed economies of scale, manufacturers of small micro systems use specialised chips to lower the chip count in their boxes. It is better to look for an appropriate machine from the low end of this market and make it into a terminal.

In the UK, the Atari ST was the first suitable system. It is very cheap and supports a high resolution bitmapped screen and a mouse. The Macintosh has never been particularly cheap in Europe. The Amiga looked interesting but it was initially priced much higher than the Atari and also the Amiga did not arrive in Europe for 9 months after the ST was launched.

The Atari ST

The first machine from Atari was the 520ST, it contains an M68000 processor and 0.5Mbyte of memory. Currently, most STs at Kent are 1040STF models having 1 Megabyte of memory. Other versions of the ST now exist with more memory. The system comes with a visually good monochrome bitmapped screen having a resolution of 640 by 400 pixels. Colour is available but is done on a different model of monitor and resolution is much lower; the current program does not support these colour STs.

For input, there is an 'intelligent' keyboard meaning that it is run by a processor which also handles time critical operations from a two button mouse. External data storage is done on 3.25" floppy. There are some ports in the back of the machine which will take a standard D-type RS232 connection, a Midi interface, a parallel printer, hard disc and extra floppies. The operating system is in ROM and mimics MSDOS at the system call level. The system which was used for most of the early development of the program read the operating system in from floppy into RAM, it took up a little under half the available memory. Having the operating system in ROM was a marked improvement.

The Atari ST has GEM as standard. GEM provides a simplified 'Mac-like' user interface with windows and pull-down menus. It is portable and exists as a programming environment on several micro systems.

Here are the compromises that must be made to achieve the right price. First, we must have a standard sized screen which is much smaller than those on most conventional workstations. Secondly, we must communicate using RS232 rather than the Ether or other fast network connection. However, in addition to the original specification, we do have a local floppy and this means that there are no worries about bootstrapping the device. It also allows tailoring information to be stored locally.

Having seen the hardware, now look for the software. There are two parts to the system: code to run in the terminal and code multiplexing shells at the host.

Host software

A keystone of UNIX philosophy is the maxim: 'Never write anything you can beg, steal or borrow'. Usenet is an inexhaustible supply of software and winging over the wires came the `uw` system written for the Macintosh by John D. Brunner.

The job of `uw` is to sit between the user processes running in each virtual terminal and the single data stream travelling along the RS232 line. `Uw` uses BSD's pseudo terminals to advantage here, each of the processes started by the user talks to their own pseudo terminal interface and data is passed through the kernel to the multiplexer. The multiplexer concentrates these several data streams into one. Since there are many windows on the screen, the multiplexer needs to support a line protocol ensuring that the data from the processes run by the user travels to and from the correct screen window. The protocol is simple, being mostly plain text interspersed with two byte control commands. Control commands are Control-A followed by a command byte. The command byte is encoded in a cunning way and allows commands like: select the window number that is to be used to receive the following data; create or destroy a window; and permits some control characters (Control-A, Control-S, Control-Q and Delete) to be sent to the user process.

The simple protocol has some advantages. First, it does not consume much line bandwidth and this is particularly important since a hidden requirement is to support a terminal in my home with the line running at 1200/75 baud. Characters come from the host at 1200 baud and are sent from the terminal at 75 baud. Secondly, the multiplexer will be left running on the host machine in a disaster such as a program crash. With `uw` it is simple to stop the multiplexer — the user types Control-A and Delete, which happens to be the command causing `uw` to die.

`Uw`'s biggest advantage is that it is available and it works. It has some disadvantages too. There is no provision in the protocol to encode window sizes, so a reshape operation on the terminal must be followed by the user running an explicit reshape command on the host for things to work nicely. The original multiplexer did not write useful information into the `/etc/utmp` file. As a consequence the processes run by the user did not integrate well into the system. For instance, many programs use the `getlogin()` routine to determine ownership and this breaks if the user's terminal is not in `/etc/utmp`. In addition, other users often attempt to use write to the user's logged-in terminal rather than a window. The code to update the `/etc/utmp` file was added into `uw` so that every window appears to be an additional user logged into the machine.

So another piece of the jigsaw is in place, we can now turn our attention to the code running in the Atari ST.

Preliminary thoughts on the terminal design

As has been said above, the ST comes with the GEM interface. At the start, it was thought that GEM would be used to provide the basic window manager for the terminal. This was ruled out because of several factors.

GEM is very screen greedy. It takes one text line across the top of the screen as a pull-down menu bar. Every window which is opened on the screen has a wide border effectively occupying one text line. This border contains a title, scroll bars and the like. GEM is slow, since portability and speed of operation are a common trade-off. It seemed that if it were possible to use lower level routines then things might go a little faster.

Another factor is that most of the commercially available C compilers for the ST are not very good. In some cases there are deplorable bugs and crucial things missing or implemented badly. In other cases, the C is 'standard K&R' and does not support many of the additions to the language such as the ability to perform structure assignment. Anyway it was better and faster to use a main-frame UNIX system as a development tool running a M68000 cross-compiler. It was therefore necessary to develop a local library of ST system calls. This was done largely 'in the dark' using the sketchy information available at the time and a considerable amount of code disassembly. However, the GEM interfaces remained somewhat impenetrable and by the time adequate information was obtained the code was well developed.

If not GEM, then what? Simon Kenyon (then at the National Software Centre, Dublin; now at ICL in Dublin) had taken Rob Pike's paper on *Layers*^{Pike1983a} and had implemented the fundamental set of

routines. Pike's layers system was implemented to support his 'Blit'^{Pike1984a} terminal so this seemed ideal.

Layers

The layers system provides a way of manipulating overlapping rectangular bitmaps displayed on a screen. Logically, the code acts as a separate module sitting between the bitmap that is the screen image and a set of application processes. The routines are designed to maintain the screen image in a way invisible to the applications code. An application knows that it possesses an area of screen and writes to it even though some of this area may be obscured by other bitmaps placed on the screen. The application has a set of special routine calls to update the screen and these routines worry about preserving screen contents.

The layers code is responsible for looking after the obscured portions of the screen. As a result, the application is not required to store any state because it is never called to redraw its portion of the screen. This works well for a simple multi-window terminal. The character comes down the line, is planted in the correct bitmap, and is immediately thrown away. This approach is attractive because it avoids potential memory allocation problems. Space for bitmaps can be carved out of free space when the user tries to set up a screen configuration; no space is needed for character storage reflecting the state of the portion of the screen. The system can never get into the position where there is memory for storing the image but no space for storing the *source* from which the image is created. Hence the system runs out of memory at predictable points rather than at some arbitrary time during the display of some important data.

The layers routines provide a basic 'system call' level in the final program. The routines deal with bitmaps and not characters so there is modularity between them and the code of the application. In addition, the routines can be developed and tested as stand-alone entities. In reality, there is no distinct applications code in every layer; all layers share the same code. Each window the user opens on the screen is a layer; a layer is really a data structure on which the code operates. The number of layers that may be opened is limited only by the amount of memory needed to store bitmaps. A major change was made to the original routines to allow the program to complain of lack of space when the user attempts to open too many layers. The code fails cleanly, losing no information, but refusing the request to create the new layer.

Some other changes are needed to the routines which Pike describes; the main one being the inclusion of a *downback()* routine giving the ability of pushing a layer to the back of the stack of layers on the screen. This is mostly used in the delete operation; the delete method described in the paper causes unpleasant screen 'flashing'.

The routines perform fairly well on the small screen. One characteristic of a small screen is that layers are overlapped more frequently than on a large screen; perhaps it is fair to say that on the small screen layers are always overlapped. The routines are good at flipping layers in and out of visibility; it is important to be able to do this with rapidity since the piece of the image that the user wants is *always* hidden behind some other bitmap on the screen.

The delete layer operation has been further altered because of this constant overlapping. When an overlapping layer is created, it causes the bitmap it obscures to be broken into several rectangular pieces. Further subdivision occurs whenever a new overlapping layer is created. The original routines never glue these rectangles back together again when a layer is deleted. This is not a great problem but can cause things to slow down appreciably in a way that is surprising to the user who does not realise the way things work. The delete operation has been modified and now attempts to rejoin these rectangles. It may fail if it cannot obtain enough memory to recreate bitmaps; in this case it leaves things as they are, hoping that a later delete operation will complete the job.

The terminal

The design of the terminal was approached with some strong ideas of how things would happen. Some of this was based on a previous terminal program written for the BBC micro†; the name *term* was also carried over from the BBC program. Much of the ideas for the user interface to the layers code were based on conference talks from Rob Pike and David Tilbrook rather than being derived from actual

† The BBC Micro is a 6502 based home computer system that was (is) much used in UK educational circles. The BBC sponsored it as part of a set of TV computer literacy programmes.

bitmapped systems. Other ideas sprang from the code development or pragmatically from the operating system running in the Atari ST.

The main features of the current program are:

1. The host communicates terminal positioning and other terminal programming information using the ANSI standard. The terminal does not emulate a VT100, although many VT100 features are supported and often programming information for a VT100 terminal will work. This was a case of 'why invent another standard when someone has created one for you'. The good thing about the standard is that it can be easily extended and bent to provide many esoteric functions. Unfortunately, it turns out to be necessary to perpetuate some of the VT100 'bugs' or 'features' because editors using *termcap* expect scroll areas to behave in certain ways. The main disadvantage with the standard is that the data sent to the terminal is a little verbose.
2. The program can support several distinct layers or areas of screen. Each layer behaves like a separate terminal with positioning information operating relative to the borders of the layer. Each layer (apart from one that fills the whole screen) has a single continuous line around it. Layers are manipulated with the aid of the pop-up menu and the mouse. Operations include create, move, reshape and delete.
3. Layers come in two types. In a *local* layer, keyboard input is simply looped back through the program directly onto the screen. In a *remote* layer, keyboard input is sent down the RS232 line to the host and data from the host is displayed on the screen. The types of layer are distinguished by having a slightly different border around them. Remote layers can be used to provide a 'many terminal down one line' capability when the multiplexer is used. Local layers are a useful notepad and can be used for temporary text storage areas in cut and paste operations.
4. In normal text mode, the terminal can display four different fonts. The Atari is equipped with 3 standard system fonts. There is a tiny 6*6 font that is too small for normal use but works well for inspecting log files in small windows and is also used by some people in menus. Term adds an extra line at the top and bottom of the font to give readable line spacing. This means that 6*6 font gives a character area of 50 rows by 106 columns. The medium sized 8*8 font gives 40 rows of 80 columns, and also has two extra lines added to the height of the font. The third font is 8*16 and is the one used by most standard terminal emulators running on the Atari ST; it produces a standard 80 column and 25 row screen. A fourth 7*8 font has been created from the 8*8 system font. This was done because I found that the 'standard' 8*8 font caused eye strain because the letter spacing was a little too wide.
Font selection is done either from a menu or under program control. Different layers may display different fonts. In addition, the terminal supports a bitmap load mode so raster images can be loaded and displayed; this was added as an afterthought to allow the examination of some bitmaps that had materialised on a tape. One user is now drawing simple diagrams using this technique — there are currently no line drawing primitives in the terminal.
5. The program normally displays black letters on a white background; this may be reversed from the menu or by program control.
6. There are some 'special' keys on the keyboard bound to specific internal functions but all other keys may be mapped into any character sequence under program control. Key bindings are loaded from the host because the code to permit specification of text strings in the terminal was likely to be too large. Key bindings have proved to not be that useful. This is largely because it is not possible to 'see' what the key contains before striking it. That being said, it is true to say that most users have some keys bound to some character sequence. Atari have different keyboard layout for each country and key binding is useful to people with non-UK Atari machines. The keys can be tailored to return the 'correct' characters.
7. A setup file may be created by the program. This contains the default settings for various aspects of the ST operating system, the RS232 initialisation information, the default output font, the default font for menus, the background colour (black or white) and the current key bindings. The setup file includes all the information normally specified by the 'Control Panel' application on the ST.

8. The program has been provided with a clock in an attempt to inhibit users from loading the processors of hosts. The clock may be loaded and started from a host or from the setup menu. This is an 'application' running in a layer or set of layers.

The use of the mouse

The ST mouse has two buttons. In general, the left button is used for selection and pointing operations while the right button summons up the menu. The mouse cursor is off unless a button is pressed, so an action is usually confirmed by releasing a button rather than pressing one. This was done to avoid the flashing caused by the need to turn the mouse cursor on and off when writing a character to the screen. Also, why have something visible until it is needed? When the mouse cursor is on, its shape is altered to show the operation which is underway.

The primary function of the left button is to select the current input layer. The button is pressed to show the mouse cursor and released when the cursor is within the boundaries of the required current input layer. The text cursor will move from the previous layer to the new one and the mouse cursor will disappear. The program thus has a strong notion of the current input layer and uses the text cursor to show the appropriate layer to the user. If the layers are overlapped, then a further left button click in the current input layer will force the image of the current layer to the front. A menu option can be used to force a layer to the back of the stack of layers.

In some programs such as editors, it is convenient to use the mouse to point at the displayed text. These programs may instruct term to go into a mode where a pointing hand is displayed. In this mode, the mouse buttons cause the program to send reports of the current text cursor position in the current layer to the host. Each button returns a different ANSI sequence and so may be used for different functions in the host program. This was inspired by the Blit support code found lurking in the jove editor; this code was altered at Kent so that the mouse could be used to mark a 'region' for the editor. Support for pointing has also been added to vi. The pointing hand code has been employed to create a menu driven host program by one user.

The Menu

The original intention was to provide different levels of menus, but in practice it has been found easier to have a single global menu containing all the operations made available to the user. The menu is displayed by pressing the right button, it 'pops-up' at the last mouse position and can thus be moved about the screen. The menu routine allows for nested menus by providing a 'sub-menu' function. Sub-menu entries are marked by a small hollow right arrow symbol on the right of the menu line. Moving the cursor over the hollow arrow symbol will cause a new menu to pop up over the first. This menu may contain further sub-menu functions. Moving outside a sub-menu will unstack it, leaving the parent menu displayed. The menu routine remembers the last selection, this is convenient for repeated actions.

The contents of the first level of menu indicate the type of operations that may be performed. The available selections are:

- New Create a new remote layer, this is replaced by 'Shell' when the multiplexer is running.
- Local Create a local layer.
- Back Put the current layer to the back.
- Copy Copy a character sequence.
- Yank Pull an area of the screen into a buffer.
- Put Put the contents of the Yank buffer into the current layer. The entry only appears when something has been Yanked.
- Put<cr> Put and append a carriage return. The entry only appears when something has been Yanked.
- Reshape Reshape the current layer.
- Move Move the current layer.
- Setup Slide right to go to the Setup sub-menu.

Delete Delete the current input layer.

The entries split into three types: entries dealing with layer manipulation; entries performing cut and paste operations with the data on the screen; and the Setup entry leading down to a set of further menus to tailor the program to the preferences of the user. Setup will not be dealt with in detail here.

The entries above are shown in the order in which they appear in the menu. This may seem a little random but it is not. The most used layer manipulation primitive is Back; the most used cut and paste operator is Copy. Since the last menu choice is remembered, the menu cursor tends to stay around this part of the menu. The Delete operator has been deliberately moved as far as possible from any other entry to avoid a button press triggering the inadvertent deletion of a layer. Just above Delete is the Setup entry; this is a useful place to park the menu selection bar after a deletion, so that a right button press has no chance of removing a layer. Reshape and Move are often done in tandem on layers and so live together. The ordering of the entries has been altered from experience of use.

Layer manipulation menu entries

The New or Shell entries allow the user to sweep out an area using a 'rubber-band' rectangle and a new layer is created in it. Normally a layer will have a single pixel width border around it; but the border will be suppressed if the layer completely fills the screen. The Shell entry creates a new remote layer and starts a shell when the multiplexer is running. Protocol limitations in uw mean that a maximum of seven multiplexed layers may be active at any one time. An attempt to exceed this will result in an error message being displayed in a specially created local layer.

The Local entry creates a new local layer. Local layers can act like a scratch pad for storing large blocks of text image, or they may be used for constructing lines of text for later insertion using either the Copy or Yank/Put functions. They do have some simple editing features. The insert key allows overwriting or insertion of characters. The delete and backspace keys do the obvious things, and in addition the arrow keys can be used to move the cursor around a character at a time. Local layers are 'free', the implementation is simply a matter of making the keyboard input routine call the routine used to place data on the screen rather than calling the routine used to send characters to the host. They have turned out to be useful and replace the tiny bits of paper kept on a desk for scribbling rough notes.

The Delete operator deletes the current input layer. The layer nearest the top becomes the current input layer. It is possible to indicate that a layer will not be used for input; so these layers will be skipped when an attempt is made to find a new current input layer. This is useful for the clock and has been used for layers running monitoring programs like *sysline*. When the multiplexer is running, the current layer can be deleted by terminating the program running in the layer. This is because the multiplexer tells the program to remove the layer using the appropriate command from the line protocol. All layers may be deleted; at which point the menu shrinks to contain Shell, Local and Setup. The command in the protocol is bi-directional. When a layer is deleted from the screen, the pseudo-terminal is closed and this results in the controlling shell being terminated.

The Back menu entry puts the current layer to the back of the current stack of layers. The layer which is now at the top becomes the current input layer. This is done because it is not easy to determine whether the original input layer is completely obscured or not.

In hindsight, the controls used to choose the layer that is to be displayed are somewhat rudimentary. There are only two: bringing a layer to the front is done by using the mouse's left button and the mouse cursor; and the current input layer can be pushed to the back with the menu selection. The lack of a 'Front' menu selection is occasionally annoying. It is sometimes necessary to fiddle around to get the screen in the correct configuration for the job in hand. This must be balanced against the temptation of making the global menu too large.

The Reshape and Move operators are self explanatory. For reshape, the mouse moves a rubber band rectangle anchored at the current top left hand corner of the current layer. Reshape is done by copying the contents of the layer into a newly created one with the new size. The original layer is deleted. For Move, a dotted rectangle shows the size of the layer and it is moved around the screen. Move is done by copying the contents of the layer into a newly created one at the new position on the screen. The original layer is deleted. Both of these operators benefit from the work done to glue bitmaps together in the delete layer

routine.

The layer manipulation operators provide a minimum set of useful things that can be done to arrange layers on the screen. There is nothing particularly revolutionary about the operations provided. Many of the operations have two or more stages and care has been taken to ensure that mouse button control is logical and consistent. For example, to create a new local layer the user presses the right mouse button to obtain the menu, moves the mouse to select Local; and releases the button. The pointing cursor, in the shape of a hollow box with an arrow pointing left and up, is displayed. The cursor is now moved to the desired position for the top left hand corner of the new layer. The left button is always used with a pointing cursor, so pressing it defines the top left hand corner. The left button is kept down while a rubber-band rectangle is displayed and the required area is swept out. Releasing the button defines the bottom right hand corner. The entire operation can be cancelled at any time by pressing the right button. The same sort of operations are needed for most of the menu choices and the same sequence of button presses and holds are used at all times. This consistency of operation is important and is one reason why the program is liked by users.

As time has gone on it has proved to be desirable to be able to carry out many of the operations from the host and ANSI sequences now exist to create new layers; put them at the front; place them at the back; reshape them and delete them. This adds greatly to the usability of the program. For instance, it is possible to execute a command on the host reshaping the layer to a certain character area.

Cut and paste operations

A main strength of any windowing environment is the ability to pick up data from one window and move it to another. A major aim of Term was to provide this functionality.

The Copy menu operator allows the user to pick up a sequence of characters from a line on the screen and insert them in the currently selected input layer at the current text cursor position. This is done by transmitting the sequence of characters up to the host and allowing the host to echo characters back onto the screen. To prevent over-running the host, provision has been made to allow the terminal to optionally delay between characters and on the transmission of a carriage return.

When Copy is selected, the left button chooses the starting position from anywhere on the screen. While the left button remains down, the mouse is moved to the right inverting the video image of the characters that are to be copied. The terminating character position can be adjusted right or left on the screen with the restriction that it cannot cross the edge of a superimposed layer. Releasing the button triggers a reverse lookup operation where the bitmap on the screen is matched with the font currently being displayed in the layer where the characters are to be found. If a match is found an ASCII character is generated. A space will be generated for any bit pattern that cannot be recognised. It is impossible to generate a tab character; the blank area on the screen will be converted into a number of spaces.

Reverse lookup is used because the program does not store the contents of a layer in any recreatable form; only the bitmap image exists. On the first call to Copy or Yank, the current font is scanned to create a lookup table. The bitmap for each character in the font is summed and this summation stored in the table. The reverse lookup code sums the bitmap for the character to be matched, rushes down the table looking for an identical value and then does a bitmap compare. Most characters in the current fonts sum to a unique number so the number of unnecessary bitmaps compared is low. The algorithm is fast enough, it can pick up a whole screen (7*8 font) in around 12 seconds, this is around 3ms per character and most of this time is spent re-inverting the video around each character being looked up.

The Yank operator is similar to Copy except that the area of the screen may span more than one line. A rectangular area of screen is picked up, a column from 15 for instance. If the selected rectangle spans more than one line, all lines but the last will have trailing spaces removed and carriage returns inserted at the end of the visible data. The data is not immediately inserted into the current input layer but held in a character buffer for later insertion using Put.

The Put entry will only appear in the menu when the Yank buffer is not empty. Selecting Put will send the current contents of the Yank buffer up the line. The Put<cr> entry will perform a Put and then send a carriage return. This is useful when a complete command is yanked; the command can be inserted into the current input layer by a single button click.

The approach of inserting copied characters at the currently active cursor position was derived from an earlier terminal program written for the BBC micro. If a user wishes to edit a line from the screen and send a slightly different version to the host then they must copy parts of the old line, type in some new characters, and then perhaps copy more sections of the old line.

Cut and paste editing is done differently on the Blit. The user moves the text cursor to the old data on the screen and edits the old image without transmitting anything. The revised image is then picked up and sent up the line. It is not clear which method is 'better'†. In Term, local layers do provide an important cut and paste tool. Large areas of the screen can be yanked and loaded into a local layer, where it may be edited before being picked up and sent up the line to the host.

Programming

The program running all this is now around 60K. Use was made of as many parts of the standard ST operating system as was possible; for instance the mouse control is all done using the standard system calls provided for that purpose. The program works hard to replace things as they were before it tailored the machine for its use; as a result it is usable in tandem with other programs written for the Atari ST. This is an issue because the machine does not support multitasking; any program can alter things in the operating system and leave an unusable machine for the next program which is run. It is particularly easy to condition the machine so that the GEM desktop is unusable, at this point all the user can do is press the reset button.

The main section is a single loop looking for events from the RS232 input buffer, the keyboard and the mouse. The appropriate routine is then called. As far as is possible, all the separate modules are called indirectly through function pointers and this allows things like local layers and switching the protocol decoding on and off to be achieved easily.

The program is constructed from a set of identifiable modules living in separate source files and having well defined interfaces. This is beginning to break down somewhat as the program has grown and operational speed became an important criteria. The main problem is the speed at which bits can be placed on the screen; this has to be faced by everyone trying to write raster graphics code. In this case, waiting for faster hardware is not an option, making the code go faster is the only choice which is available. This is not quite true, Atari has long promised a special blitter chip and the code uses the standard Atari blitter interface and *bitblt()* routine to move bits around memory in anticipation that the chip will be made generally available.

However, the program uses several techniques to make things work a little faster. The first thing to notice is that writing characters to the screen is slow, it is not possible drive the screen at 9600 baud when using the RS232 connection running at that speed. This means that the code can refrain from updating the screen if there is anything in the RS232 input buffer. This observation can be used to make things work much faster by avoiding unnecessary work. For instance, to write a single character on the screen means that the text cursor (which is an video inverted character area) has to be turned off; the character loaded from the font onto the screen (possibly using the layers routines) and finally the text cursor turned on again. This trick here is to ensure that the program does not turn the text cursor on again if there is anything in the RS232 input buffer.

Another win with this technique is to build up a line in an off screen bitmap using the standard *bitblt* code; the entire line is then pasted onto the screen using the layers version of the *bitblt* routine. Since the layers code calls the *bitblt* routine, this saves much unnecessary work.

Scrolling is the slowest operation that has to be undertaken; it mostly means that nearly all the screen has to be moved up or down memory. The worst case is scrolling nearly the entire screen bitmap when moving the whole screen up one line. Many terminals perform scrolling by moving the base address loaded into the video control chip. This method is not available on the Atari ST; scrolling simply has to be done by physically moving large chunks of memory around. However, things can be made to go a little faster if blank areas are to be displayed. The program simply counts newlines when lines are blank and unloads this cache when the RS232 input buffer empties; this results in jump scrolls of more than one text

† A religious issue, I suspect. Depending on which method the user came across first.

line. It is particularly useful when inside a visual editor using scrolls to clear portions of the screen; this now happens very quickly.

Another speed-up can be obtained by noticing that an operation is to be done on a full screen layer that is totally visible. Here the layers code is not needed and can be avoided, also scrolling can be performed quickly using byte copying routines that are much faster than the *bitblt* code because they are moving aligned data. This is slightly nasty but is justifiable because many people use the program when displaying a single unobscured layer.

Possibly things cannot be made to go much faster, a compiling *bitblt* routine might help or perhaps Atari might get round to selling the long promised chip. However, it is unlikely that the program will be developed much further since it has reached that stage of complexity where something new should be done to replace it.

Conclusion

The Term program provides a windowing environment for software development running on commercial hardware bought for a fraction of the cost of a workstation. The system appears to be well liked by users whose main problems stem from the hardware — ‘the screen isn’t big enough’ or ‘it needs to run faster’. All the users previously had a text terminal on their desk and find that the new system significantly adds to their working environment. It is unclear whether workstations will become available to all who wish to use them, we all need much cheaper large screens and network hardware for this to become a reality. In interim, cheap systems like Term can provide much of the functionality at a price that more people can afford.

References

Pike1983a.

Rob Pike, “Graphics in Overlapping Bitmap Layers,” *ACM Trans. Gr.*, vol. 2, no. 2, pp. 135-160, April 1983.

Pike1984a.

Rob Pike, “The Blit: A Multiplexed Graphics Terminal,” *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1607-1631, October 1984.

GATED: A MULTI-ROUTING PROTOCOL DAEMON FOR UNIX*

*Mark S. Fedor
NYSERNet Incorporated
Rensselaer Technology Park
125 Jordan Road
Troy, New York 12180*

fedor@nisc.nyser.net

Abstract

With the emergence of the NSFNET backbone network and its innovative routing protocol, it became necessary to develop a way to interact with the backbone network's routing strategy. Also, during this time, many mid-level networks were becoming connected to the NSFNET backbone. Individual sites on these mid-level networks wished to communicate via the NSFNET backbone, but also wanted to use their own DDN ARPANET or MILNET link as a fallback route when necessary. Of course, this use of backup and fallback routes had to take place dynamically in order to be of any real value.

This paper discusses the implementation of a UNIX routing daemon which is capable of performing the above described tasks as well as a number of other functions. Experiences in dealing with routing protocols when they are running in parallel and how they affected design decisions will also be discussed. This routing daemon has been in the field for almost a year and a half. While at first it was believed to be a very short term solution, it has proven itself well and has given us the necessary time to push forward towards the development of a new routing protocol.

1.0 History and Background

The NSFNet backbone became operational during June of 1986. This backbone network consisted of six nodes, one at each of the NSF funded supercomputer centers. Each node consisted of a PDP 11/73 (fuzzball) gateway running software (fuzzware) developed by Dave Mills of the University of Delaware [1]. The routing protocol used by the fuzzball is very different from the routing protocol used widely within UNIX environments. The fuzzball uses

* UNIX is a trademark of AT&T Bell Labs

the Hello routing protocol, while the most widely used routing protocol within UNIX environments is the Routing Information Protocol (RIP).

The Hello routing protocol has a metric based on time delay. Its metric has a range of 0 to 30000 milliseconds with 30000 representing infinity or unreachable. The Hello protocol also has a number of features which add to the stability of routing and suppress problems such as routing loops and metric counting. These features include the performance of split-horizon, hold-downs, and flash updates [2].

Split-horizon can be defined as using the infinity metric in the routing information you send out for routes that are being gatewayed through the interface you are sending the routing information on. In less confusing terms, you don't broadcast reachability for a network back out of the same interface you are reaching that same network on.

Performing hold-downs means that once an infinity metric is heard for a route, it is not deleted, but kept at infinity for a given amount of time. During this hold-down period, no routing information is believed about the route being held-down. The gateway itself continues to announce the held-down route during the hold-down period. This allows the entire system to propagate the unreachable network before it is actually deleted.

Performing flash-updates means that once a route has changed, propagate the change immediately. Don't wait for the usual interval for sending a routing update. This allows networks to react quickly to changes.

The RIP used in conjunction with BSD UNIX is a derivative of the Xerox Routing Information Protocol [3]. It has a metric based on hop counts and the range is from 0 to 16 with 16 being infinity or unreachable. RIP, as it appears in BSD UNIX, was not intended to be used in wide area network applications and therefore has no routing features to promote stability as the fuzzball does. As we will discuss later, using RIP in a wide area network environment brought on a number of problems.

As soon as the NSFNet backbone became operational, the technical people involved in operating and developing the NSFNet backbone realized that in order for sites to make use of the backbone effectively, some routing agent had to be developed to interact with the backbone using Hello, while also interacting with connecting sites using RIP and EGP. This routing agent would mediate and translate between the protocols, passing information from one protocol to the other and back. While manually adding static routes may have been easier in the short term, it would not have fully utilized the redundancy now present in the Internet environment. Using static routes would have made users of the backbone "handicapped"; so full priority was given towards making the NSFNet backbone a fully integrated part of the Internet. This routing agent, *gated*, was to be a big step in fulfilling these plans.

The development of *gated* started in August of 1986 and with input from such people as Dave Mills, Mike Petry of Maryland, Scott Brim of Cornell, and Hans-Werner Braun of Merit, the first release of *gated* was available in November of 1986. Since then, there have been three more releases. Each release contained fixes and enhancements which were made after gaining

experience in working with the previous release. Development of gated continues even now, changing and improving as the needs of the Internet world continue to grow.

2.0 Overview of Gated

The gated routing daemon for UNIX handles two different routing protocols, Hello and RIP, and one reachability protocol, EGP [4]. Gated can participate in any combination of the three routing protocols at one time. It is meant as a replacement for the UNIX routing daemon *routed*, the UNIX EGP daemon, *egpup*, and any sort of Hello routing daemon that may be used.

In addition, gated listens to ICMP redirects and attempts to deal with them in a rational manner [5]. Gated also provides routing configuration options which make developing a network routing strategy much easier for the network administrator. These routing options include listening to certain listed gateways for routing information, only putting certain listed networks in the routing update packets, turning off routing on a given interface on a per-protocol basis, and the sending of routing information directly to listed gateways.

Gated keeps its own internal routing tables. At startup, gated reads all of the routes out of the UNIX kernel to get an initial state. Once this initial state is achieved, gated then dictates what routes are entered into the kernel. Since the routing process in UNIX is out of the kernel, the kernel has to mirror the routing process which is actually an application level program. The only time in which gated tries to parallel the kernel is when an ICMP redirect is received by the UNIX gateway. The ICMP redirect has already been acted upon when the kernel passes it up to the application level. Gated must then try to make the same actions as the kernel in order that each of their routing tables remains consistent with each other.

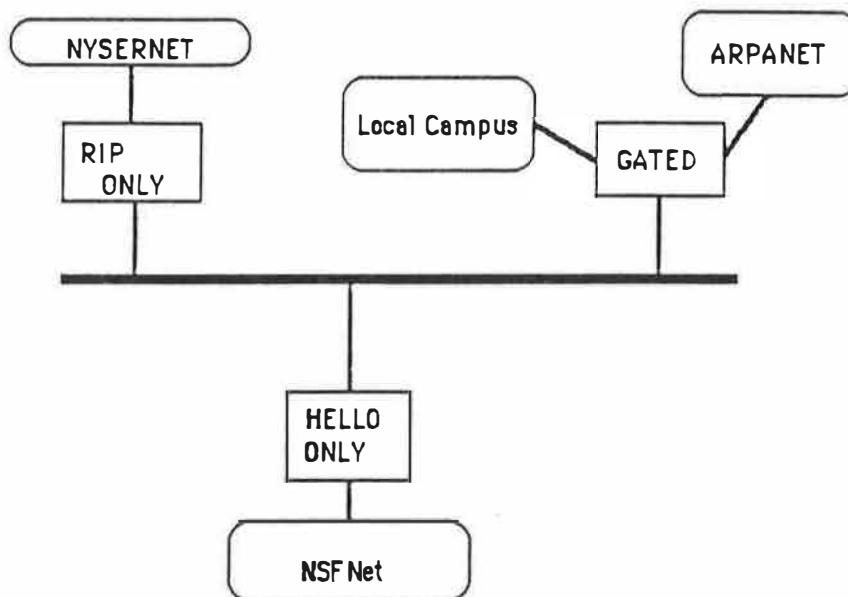


Figure 1. Typical Gated Environment

Figure 1 shows a topology in which gated is most widely used. As you can see, the only way to get routing information from NYSErNet to the NSFNet is to have a mediator do some routing translation between RIP and Hello. The gated machine above can do that. Also, the gated machine will prefer routes coming from the NSFNet as opposed to routes coming from the ARPANET. If the NSFNet gateway goes down, the gated machine will fall back to using the ARPANET. This is explained in detail in a later section.

3.0 Gated Modules

The following sections describe the different modules of the gated implementation. Because gated is an application level program, additional fields were added to the routing table structures and interface structures without having to make any kernel modifications. These additions made the processing of routes and interfaces much easier. These additions will be discussed in detail below.

3.1 RIP Implementation

The RIP implemented in gated is the same that was implemented in routed for BSD UNIX with a few minor additions [6]. The RIP in gated includes the routing features of the Hello routing protocol. These features are split-horizon, hold-downs, and flash-updates. These features were not included in the BSD UNIX RIP implementation because RIP was not meant to be used in a wide area network. In order for RIP to interact with the NSFNet routing scheme, these features were needed. The split-horizon, hold-downs, and flash-updates in gated's RIP parallel the implementations found in the NSFNet fuzzball gateways. The RIP hold-down period in gated is one minute. RIP packets are supplied gratuitously every 30 seconds and RIP routes are timed out after four minutes of not being updated. On startup, gated sends out an initial RIP command request packet asking all gateways for RIP information.

All of the RIP routing features are subject to the configurable routing options and firewalls of gated. For example, a flash-update will not be sent via RIP for a network that can only be announced via the Hello protocol. Hold-downs, coupled with split-horizon, effectively cut down on "counting to infinity" problems associated with RIP. "Counting to infinity" is an event which occurs when connectivity to a network is lost. Gateways participating in routing who do not employ these special routing features each claim false connectivity to this network. Each time they claim connectivity, the routing metric used is higher than the last one used. This condition prevails among the gateways until each one of them uses the infinity metric. Once this condition is reached, the information about the lost network goes away, but while the gateways were "counting" the route and claiming false connectivity, all network traffic destined to this network was lost unknown to the users of the network.

3.2 Hello Implementation

The Hello implementation in gated parallels the fuzzball gateway of Dave Mills as closely as UNIX would allow it. Specifically, gated follows RFC 891 and has attempted to keep up with the modifications to the Hello protocol which were made by Dave Mills as the needs of the NSFNet backbone changed.

The native Hello protocol requires a very accurate clock. The PDP-11 fuzzball system provides the accurate timing needed by the Hello protocol. UNIX could not provide the accuracy needed for full participation in the Hello protocol with the fuzzball, but participates in the Hello protocol without affecting the timing of the fuzzballs by invalidating the time stamp in the Hello packet header.

The Hello protocol has all of the routing features described in the previous section and is subject to the same firewalls. A Hello update is generated gratuitously every fifteen seconds. The hold-down period is one minute long. The Hello protocol in gated does not include subnets, only the non-subnetted network number.

3.3 EGP Implementation

The gated EGP implementation is the original Kirton EGP application with some bug fixes and slight enhancements [7]. The firewalls and routing options were also applied to EGP. Control of what dynamically went into the EGP updates was now possible. Considerable work was done by Thomas Narten of Purdue to improve the timing modules of the EGP implementation. Narten also modified gated to peer with the new BBN Butterfly gateways to be used in the ARPANET core. The gated EGP implementation was modified to allow two implementations to peer with each other in a subnetted environment. This feature was used at the John Von Neuman Center in Princeton, N.J.

3.4 ICMP

Unlike its predecessors, gated listens to ICMP redirects and figures them into the routing scheme. Once gated receives a redirect, it attempts to mirror the kernel with its own actions. Any route learned of via an ICMP redirect is not broadcast via any routing protocol and is deleted from the kernel after six minutes. This is different from the kernel as it will never remove a route installed via an ICMP redirect. The ICMP processing by gated has tended to make the routing tables much cleaner and more consistent. Before gated, it was not uncommon to see as many as ten different routes for the same network after a flurry of redirects.

3.5 Interface Processing

Interface information is read from the kernel during startup. Only those interfaces that have been ifconfig'ed before gated startup are considered. An interface that is down is noted as such and is periodically checked by gated (every 30 seconds). If the Interface flags have changed since the last 30 second check, the new flags are put into gated's internal interface list and the appropriate actions are taken. For example, if an interface goes from down to up, gated will read the address, netmask, and all other information needed to reinitialize the interface structure. Therefore, if a change of the broadcast address or interface metric of an interface is needed and gated can not be restarted, ifconfig the interface down until gated notices it and ifconfig it back up again. The change will then take effect in gated.

After interface list initialization, gated initializes its own routing tables with the direct routes associated with each interface. These routes are not

added to the kernel routing tables as the *ifconfig* command has already done this. Gated installs the interface routes with a metric of zero and the route type as "direct". The interface routes are not marked as passive (static) and do indeed age. Every time a routing packet is received on an interface, the direct route associated with that interface has its age reset. If an interface route should age and have to be deleted, the route is deleted from the kernel as well as gated's own tables after the hold down period. From this point on, gated will then listen to any other information regarding its former direct interface route. While gated has a direct interface route, it will ignore any other routing information received about the interface route. If gated should receive a routing packet from the previously down interface, it will reinstall the direct interface route and delete any other route associated with this destination. This feature only applies if there are two or more interfaces and the interface is not specified in a "passiveinterface" clause in the configuration file.

A link to a PSN or IMP is tested by sending a routing packet to itself. The packet will go out to the IMP and get sent back. On receipt of this packet, the age is reset on the direct interface route associated with the IMP.

Gated's internal interface structure is described in greater detail in Appendix A. The interface structure is defined using the C programming language and is an excerpt from a gated source header file.

3.6 Route Processing

Gated keeps a single internal routing table. The table is hashed as in the 4.3BSD UNIX kernel. The hash function is also the one that is used in the UNIX kernel. Each of gated's routing table entries contains an abundance of information about the route itself. This is much more information than the UNIX kernel holds, allowing gated to perform all of its routing features rather easily. The gated routing entry is described in detail under Appendix B. The route entry structure is defined using the C programming language and is an excerpt from a gated source header file.

Each destination network that gated chooses to place in the routing tables has an entry as described in Appendix B. The initialization of the entry for the destination network and the actual placement into the routing table takes place after gated makes a decision whether the route to this network is actually better than any other route it has for this same network. This decision making process can get very complicated and is discussed in section 5.0 on routing interchange.

3.7 Firewall Processing

Gated's firewall system provides configurability as to where and how routing information about a given destination network will be given out. Gated also provides the same feature for listening to information about a given destination network via the routing protocols. Configuration of the firewall system consists of adding entries to a configuration file. At startup, gated reads these entries and keeps a static list of destination networks and the restriction information for each. When a route for a destination network is

added to the gated internal routing table, a search is made of this restriction list to check for any restrictions on this network. If any are found, a field in the routing table entry points to the proper restriction information to make it readily available for lookup.

4.0 Routing Features

The routing features supported by gated are those described in section 3.1. These features help in the suppression of routing loops and "counting to infinity." "Counting to infinity" is a real problem when you have a very large redundant network. The fuzzball routing algorithm has gone to great lengths to suppress these problems. Therefore, the gated routing algorithm has attempted to parallel the fuzzball's features as closely as possible. It was also necessary to interact with the fuzzballs without affecting their routing performance. This was a real challenge.

In applying these routing features, gated had to take into account the possibility that these protocols might run in parallel on the same network. For example, on one network, there might be three gateways running Hello and RIP, one gateway running just RIP, and one fuzzball running only Hello. If split-horizon is applied to all routes straight across the board no matter what routing protocol they were heard from, certain routes that were heard from the RIP only gateway by the gated machine would never be told to the Hello only fuzzball. The gated machine would apply split-horizon to the RIP only route and Hello it out at an infinity metric. The fuzzball would then of course ignore this. This would totally defeat the original purpose of gated, which was to be a routing agent/translator between RIP and Hello. To solve this problem, gated keeps a per-interface list of participating routing gateways and what protocols each is speaking. This list is kept in the interface structure. Furthermore, each routing table entry contains a flag that tells what routing protocols the next hop gateway is speaking. When it comes time to send out a routing update via a routing protocol (N), gated checks to see what the next hop gateway is speaking for each route. If it is speaking the same routing protocol as protocol (N) and the next hop gateway is out the same interface as the protocol (N) update packet is going to be sent, set the metric to infinity. This is gated's split-horizon strategy in a parallel routing protocol environment. Split-horizon is also not applied to any static routes.

5.0 Routing Interchange

When gated must make a decision on whether to pick one route over another, it can become rather complicated. The easiest method is, obviously, to pick the route with the smallest metric. In the multi-routing protocol environment, this isn't always the case. Gated must take into account the possibilities of different metrics and interaction problems between the routing protocols.

The two routing protocols that gated supports, RIP and Hello, both have different metrics and have some interaction problems that affect the two of them when operating in parallel. One of the interaction problems has to do with the metric conversions between both protocols. When a metric is converted from one protocol to another and then back to the original protocol, the metric must be consistently incremented so the integrity of the metric is maintained. Internally, gated keeps its metrics in milliseconds. This is so the

granularity of the Hello metric is preserved. If the Hello metric were to be converted to a hopcount metric, its usefulness in making fine-toothed routing decisions would be severely limited. Therefore, the RIP hopcount metric is converted to milliseconds. In order to accomplish this a conversion table is used. The conversion table used in gated version 1.3.1 is listed in Appendix C.

In making route decisions, any Interior Gateway Protocol (IGP) learned route (Hello or RIP) is always preferred over an EGP learned route. This means that any ARPANET/MILNET learned route will not be used if there is a path to that route via Hello or RIP. As soon as the IGP route is deleted, the EGP route will be used if it is still available. Since the ARPANET /MILNET use an EGP and the NSFNet backbone uses an IGP, any gateway running gated will use the NSFNet backbone routes and use the ARPANET/MILNET as a fallback for the NSFNet routes. The reason for having the IGP unconditionally replace the EGP is because the EGP used in the Internet today is a reachability protocol and although a metric is carried through, it can not be trusted. This means that it cannot be used to make routing decisions. There are future plans to upgrade the EGP to carry valid metrics through the system. Once this is done, gated can be enhanced to have the EGP and the IGP's interact more intelligently.

A static route is never overridden by a routing protocol learned route, and is also never aged. It is kept in the tables until the configuration file is changed and gated is restarted. A RIP learned route does not override a Hello route unless the RIP route is better than the minimum of the last 24 Hello metrics for the route in question. This is to guard against RIP implementations that do not perform split-horizon and the momentary undercutting of a Hello route if the Hello metric suddenly changes by a wide margin. The last phenomenon was experienced during the initial months of NSFNet backbone operation.

6.0 Routing Configurability

As was mentioned earlier, gated allows flexible routing configurability. In addition to the firewalls, gated allows the specification of static routes, per-interface routing protocol control, the setting of interface metrics, the initial setting of a default route, and many more that cannot be discussed in this paper. For a complete description of the configuration options for gated, see the gated UNIX manual page in the gated distribution [8].

7.0 Future Developments

Gated is continuing to be developed at the Cornell Theory Center. Since I departed Cornell in November of 1987, gated development has been taken on by Jeffrey C. Honig. For the most part, gated has implemented all of the enhancements to the Hello protocol as done by Dave Mills since that time and the support for fixed metrics has also been added. Gated is being modified to be used in the new NSFNet backbone network being deployed by IBM and Merit, Inc. The modifications include the addition of a Shortest Path First (SPF) routing protocol [9] and further enhancements to the EGP. More routing configuration options are being added to aid the network administrator in interfacing into the new NSFNet. The feedback from the Internet community has been excellent and has been one of the reasons for the success of gated.

Initially, gated was developed as an interim measure, but will continue to be developed as long as it is needed in the community.

8.0 Acknowledgements

I would like to acknowledge certain people for playing a large role in the development of gated. Most notably, Scott Brim and the Cornell networking crew for helping me hash out problems; Mike Petry of Maryland, Dave Mills of Delaware, and Hans-Werner Braun of Merit for giving me much advice on gated development; all of the NSFNet technical crew who provided invaluable feedback; and the beta-testers who did their job well. It takes more than one person to develop a program that is useful to the majority of the Internet community.

9.0 References

- [1] Braun, Hans-Werner and Mills, David L. The NSFNet Backbone Network. Proceedings SIGCOMM 87. Stowe, Vermont, August, 1987.
- [2] Mills, David L. DCN Local-Network Protocols (Hello). Network Working Group Request for Comments, RFC-891. Network Information Center, SRI International, Menlo Park, California, December 1983.
- [3] Routed(8) - RIP routing daemon. UNIX Manual description pages, University of California at Berkeley, April, 1986.
- [4] Mills, David L. Exterior Gateway Protocol Formal Specification (EGP). Network Working Group Request for Comments, RFC-904. Network Information Center, SRI International, Menlo Park, California, April, 1984.
- [5] Postel, Jon Internet Control Message Protocol (ICMP). Network Working Group Request for Comments, RFC-792. Network Information Center, SRI International, Menlo Park, California, September, 1981.
- [6] Hedrick, Charles Routing Information Protocol (RIP). Short-Term Routing Taskforce, Internet Engineering Task Force, Draft RFC, Rutgers University, March, 1988.
- [7] Kirton, Paul EGP Gateway under Berkeley UNIX 4.2. Network Working Group Request for Comments, RFC-911. Network Information Center, SRI International, Menlo Park, California, August, 1984.
- [8] Fedor, Mark S. Gated - Network Routing Daemon for UNIX. UNIX Manual Description pages, Cornell University, December, 1986.
- [9] Rekhter, Jacob EGP and Policy Based Routing in the New NSFNET Backbone. T. J. Watson Research Center, IBM Corporation, Yorktown, New York, March 6, 1988.

Appendix A: Interface Structure

The interface structure definition was taken from the header file "if.h" of the gated version 1.3.1 distribution package.

```

struct interface {

    struct interface    *int_next;
    struct sockaddr     int_addr;

    union {
        struct sockaddr    intu_broadaddr;
        struct dstaddr     intu_dstaddr;
    } int_intu;

    u_long              int_net;
    u_long              int_netmask;
    u_long              int_subnet;
    u_long              int_subnetmask;
    int                 int_metric;
    int                 int_flags;
    int                 int_ipackets;
    int                 int_opackets;
    char                *int_name;
    u_short             int_transitions;
    struct active_gw    *int_active_gw;
    int                 int_reconstmetric;
}

```

<i>int_next</i> -	A pointer to the next interface in the list.
<i>int_addr</i> -	The address of the interface on this gateway.
<i>int_intu</i> -	A union which contains either a broadcast address or destination address depending on whichever is applicable.
<i>int_net</i> -	Network number of the interface.
<i>int_netmask</i> -	The network mask of the interface.
<i>int_subnet</i> -	The subnet number if applicable.
<i>int_subnetmask</i> -	The subnet mask of the interface if applicable.
<i>int_metric</i> -	The interface metric.
<i>int_flags</i> -	Marks state of interface: up, point-to-point, etc.
<i>int_ipackets</i> -	Number of routing packets received.
<i>int_opackets</i> -	Number of routing packets sent out.
<i>int_name</i> -	Interface name in UNIX.
<i>int_transitions</i> -	Number of times interface went from down to up.
<i>int_active_gw</i> -	List of other routing gateways on the interface.
<i>int_reconstmetric</i> -	Metric for reconstitution of route metrics.

Appendix B: Routing Entry Structure

This structure represents an entry in gated's routing table. As discussed earlier, this entry holds more information than the UNIX kernel's routing entry. This increased knowledge allows gated to perform its complex route manipulations.

```

struct rt_entry {

    struct rt_entry    *rt_forw;
    struct rt_entry    *rt_back;

    union {
        struct rtentry    rtu_rt;
        struct {
            u_long        rtu_hash;
            struct sockaddr    rtu_dst;
            struct sockaddr    rtu_router;
            short          rtu_flags;
            int            rtu_state;
            int            rtu_timer;
            int            rtu_metric;
            struct interface    *rtu_ifp;
            int            rtu_proto;
            struct hello_win    rtu_h_window;
            struct restrictlist    *rtu_announce;
            struct restrictlist    *rtu_noannounce;
            struct restrictlist    *rtu_listen;
            struct restrictlist    *rtu_srclisten;
            int            rtu_fromproto;
        } rtu_entry;
    } rt_rtu;
};

```

<i>rtu_dst</i> -	The destination network.
<i>rtu_router</i> -	The next hop gateway for the destination network.
<i>rtu_flags</i> -	Marks if route is direct, dynamic, host, or up.
<i>rtu_state</i> -	Marks state of route: just changed, passive, etc.
<i>rtu_timer</i> -	The age of the route.
<i>rtu_ifp</i> -	Pointer to the interface from which the next hop gateway is reachable.
<i>rtu_proto</i> -	The routing protocol from which the route was learned.
<i>rtu_h_window</i> -	Keeps the last 24 Hello metrics for the destination network.
<i>rtu_announce</i> -	Pointer to information about what gated can announce and where.
<i>rtu_noannounce</i> -	Pointer to information about what gated can not announce.
<i>rtu_listen</i> -	Pointer to information about what gated can believe about this route including what protocol and what interface.
<i>rtu_srclisten</i> -	Pointer to information about what gated will believe about this route.
<i>rtu_fromproto</i> -	Flags what routing protocol(s) the next hop gateway is speaking.

Appendix C: Metric Conversion Table

This Appendix outlines the metric conversion table in gated version 1.3.1. The table was provided by David L. Mills of the University of Delaware. Please note that once a RIP metric is converted to Hello milliseconds, any addition of milliseconds to the converted metric guarantees a higher RIP metric when converting back to a RIP metric. This property is needed so that routing loops don't occur and the integrity of the metric is somewhat maintained.

<u>RIP</u> <u>Hopcount</u>	<u>Hello</u> <u>Milliseconds</u>	<u>RIP</u> <u>Hopcount</u>
0	0	0
1	100	1
2	148	2
3	219	3
4	325	4
5	481	5
6	713	6
7	1057	7
8	1567	8
9	2322	9
10	3440	10
11	5097	11
12	7552	12
13	11190	13
14	16579	14
15	24564	15
16	30000	16

Asmodeus

A Daemon Servant for the System Administrator

Mark E. Epstein
Curt Vandetta
John Sechrest

Department of Computer Science
Oregon State University
Corvallis, OR 97331

{epsteinm,curtv,sechrest}@cs.orst.edu

Abstract

Asmodeus is a distributed database system which simplifies the tasks of the Unix systems administrator. Using simple commands to the Asmodeus command interface, the systems administrator can perform many tedious operations over a large network of machines. The system uses a reliable datagram socket connection for all of its communication. It is designed to withstand server crashes so that the average user never knows when it happens. Asmodeus synchronizes changes across machine boundaries, while performing many varied tasks including changing passwords, adding and removing user accounts, and adding and removing users from groups and mail aliases. Programs which only read standard Unix system databases need not be changed. Asmodeus operates behind the scenes, uses minimal system and network resources, and can be used with binary-only Unix systems.

1 Introduction

The operational problems that Oregon State University's Computer Science Department is faced with are common. Our world is a network of many different types of Unix¹ systems (including BSD 4.3, DYNIX 3.0, IIP-UX 5.5, and UTek 2.4 systems). We have few system administrators, and a large, frequently changing user base. In concert with the cumbersome system administration tools provided with most Unix systems, our environment causes inconsistency problems between machines, and the few system administrators we have spend too much of their time fixing them. In this paper, we discuss these problems and our solutions.

A typical task which uses much of the administrator's time involves changing a person's username so that it is consistent across our systems, or more to his/her liking. To change

¹ Unix is a registered trademark of AT&T Corporation

a username, the administrator must perform many exacting steps, which takes about five minutes of the administrator's time for **each** name change. If this were not enough, when the administrator forgets just one of the steps, it leads to blatant failure.

In the Asmodeus system, we have designed a distributed database system which can relieve the system administrator of much drudge-work, by remembering and executing all of the steps in the change-of-username process and in many other processes, including (but not limited to) adding and removing user accounts, and adding and removing users from groups and mail aliases. Asmodeus also can support some root-level file access on remote machines, without requiring a login session. Using the Asmodeus system for tasks it supports is much simpler and faster than using existing tools, increasing the administrator's productivity and allowing him/her to focus on the important aspects of the job. As well as freeing the administrator from tedium, Asmodeus also supplies information enabling the administrator to track the use and abuse of the systems he/she administers.

The sections of this paper are: a journey through the Asmodeus system; the database daemon (DBD); the activity daemon (ACTD); the generic daemon (GENERICD) (a daemon-builder's kit); and the Asmodeus communications protocol (RDCP). Also covered in this paper are the differences and similarities between Asmodeus and YP [SUN86{a,b,c}].

2 A Journey through Asmodeus: (The Global View)

Consider a hypothetical user `jones` who has an account on a machine named `jacobs`. When `jones` wants to change her password, she runs the `passwd` program. `Passwd` follows these steps in changing the password file:

- `Passwd` asks `jones` for her old password,
- Encrypts the password, compares it to the one stored in the `/etc/passwd` file, and
- If it matches, asks `jones` for her new password,
- Encrypts the new password and puts it into the `/etc/passwd` file.

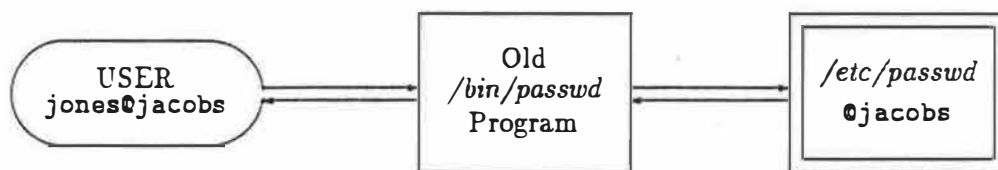


Figure 1: Original `/bin/passwd` Program

We have rewritten `passwd` (and some other standard UNIX programs) to use RDCP, because they cannot be used with Asmodeus without doing so. Programs which **write** on standard UNIX databases must be rewritten to be used with Asmodeus. Externally, the modified

passwd program works exactly the same as the original, but the internal workings are very different, as shown in this list the steps followed by the Asmodeus *passwd* program:

- *Passwd* uses the Reliable Datagram Communications Protocol (RDCP) library to start a conversation with the Database Daemon (DBD) (possibly on some other machine),
- Requests the DBD to send to *passwd* its copy of *jones*' encrypted password,
- Asks *jones* for her old password,
- Encrypts the password, compares it to the one received from the DBD, and
- If it matches, asks *jones* for her new password,
- Encrypts the new password, and sends it to the DBD.

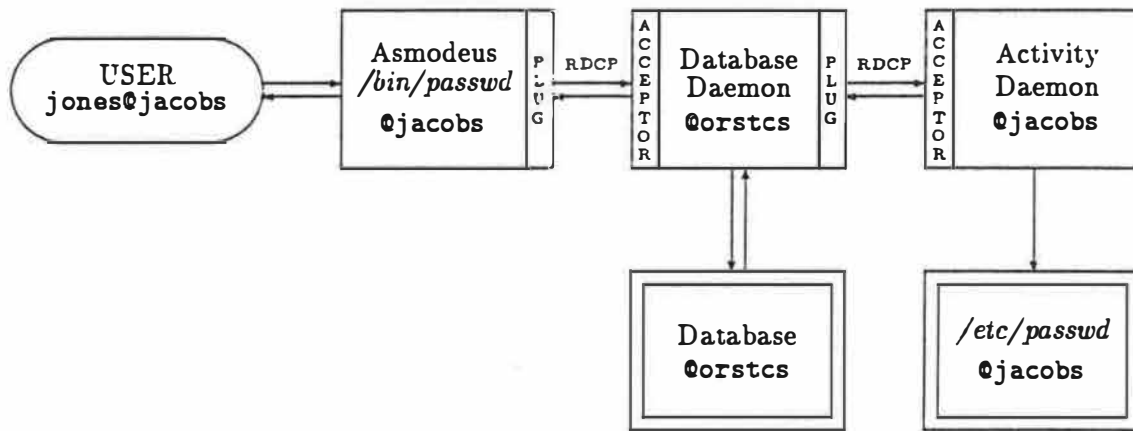


Figure 2: A Simplified View of Asmodeus

The Asmodeus *passwd* program need know nothing at all about the format or location of the password file, but instead sends the DBD the new encrypted password. The interpolation of the DBD and ACTD between the *passwd* program and the */etc/passwd* file is the key element which allows Asmodeus to track and report changes to the group of systems which it services. In the example given, the actual */etc/passwd* file on the machine has not changed yet. That happens next.

When the DBD receives the command to change a password, it changes its database immediately, and also generates a command for the ACTD running on the particular machine which issued the change request. When the DBD sends the command to the ACTD, the ACTD changes the */etc/passwd* file. If *jones* had instead used the *chfn* program, the DBD would have sent a command to change *jones*' full-name field to the ACTD of every machine on which *jones* had an account. Each type of change request sent to the DBD generates the correct message(s) to the appropriate ACTD(s).

3 Database Daemon

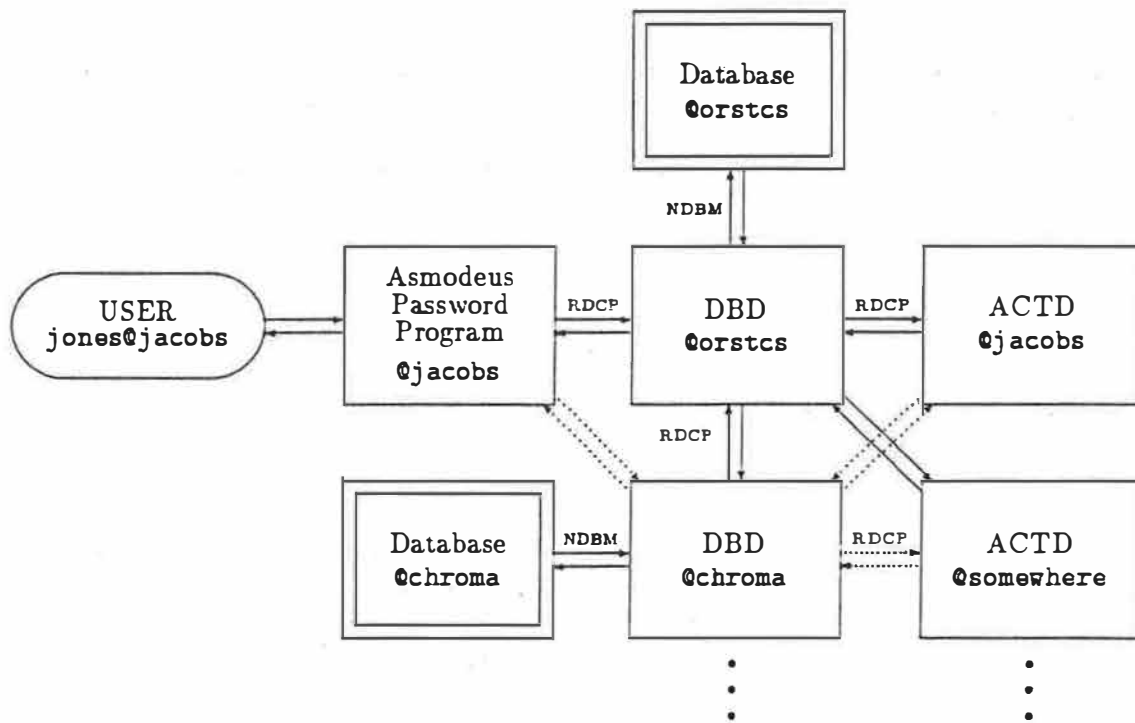


Figure 4: The Full Asmodeus Picture

3.1 Introduction to the DBD

The DBD is the only process in the Asmodeus system which is both a server and a client, and, as such, fills a special role as the Asmodeus command translator and rule enforcer. It is a server to client programs like *passwd*, and a client to the Activity Daemon (Section 4). Actually, the DBD is not a singular entity; it is a collection of DBD processes, each running on a separate machine, which continually communicate update information to one another.

All of the intelligence of the Asmodeus system is bundled into the DBD. The DBD does the following:

- Services client to database requests.
- Understands the relation of the database to the ACTD(s), and sends appropriate requests to the ACTD(s).
- Sends update information to other DBDs.
- Logs all transactions.

3.2 Some Basic Functions

In Section 2, we discussed what happens in the Asmodeus system when `jones@jacobs` wants to change her password. The messages that are sent by the *passwd* program to the DBD are:

```
open
lock jones
read Login_name jones Machine_name jacobs Password
change Login_name jones Machine_name jacobs Password LKprau6SUwhZo
unlock jones
close
```

This is a small but representative fraction of the commands which client programs can send to the DBD. The full list of functions can be found in appendix A. The client to DBD conversation would be similar had a system administrator run the *update* program and added `jones` to a group or mail-alias, instead of `jones` using the *passwd* program to change her password.

3.3 Conversion of Database Changes to ACTD Commands

While *passwd* is talking with the DBD, the DBD accumulates information which it will send to `jacobs`' ACTD, when it is convenient. When the DBD is not too busy, it opens a connection to `jacobs`' ACTD and sends it the commands (Section 4) required to change the password entry for `jones` in `jacobs`' */etc/passwd* file. The DBD knows what ACTD command(s) to generate for each modify request it receives.

3.4 Redundancy Features

When a client process wants DBD service, it calls an Asmodeus function which talks to local machines to see if they are running a DBD. If it finds one, then that is the DBD which the client will communicate with; if it does not find a DBD, then presumably there are no DBDs running which can service the client's requests.

There are two basic types of database manipulations that Asmodeus supports: query and modify. Every time a DBD receives a query request, it services that request **immediately**, using the information in its local database, unless it knows that its knowledge of the data item could be out of date. Thus most queries (and most accesses to this database **will be** queries) can be serviced quickly.

When a client process wants to **change** data stored by the DBD, it first must request a lock on the data item. In the previous example, the *passwd* program requested a lock on the `jones` record. When a DBD receives a lock request, it forwards the request to other DBDs. If it does not get a refusal from any DBD, then it sends a lock confirmation to all of the DBDs, and grants the lock to the client program.

This locking mechanism is how a DBD would know that its knowledge of a data item could be out of date. If a DBD receives a query request regarding a data item which could be out of date in its local database, it will forward the request to the DBD which locked the data, and then forward the answer back to the client program. This forwarding is necessary to ensure that the client programs always have the same idea of the current contents of the (distributed) database.

A client program may not change any Asmodeus data without first being granted a lock on a record. Then the client can change any writable data in the record which it has locked. It must unlock the record when it is done. When the client process requests to unlock a record, its DBD grants the request immediately, then makes sure that all other DBDs are correctly updated before releasing its lock on the data. This ensures data consistency.

While a client process has a lock on some data, it can send modify requests to its DBD. After each modify command is received by the DBD, **before** the DBD sends an OK message to its client, it forwards the change request to another DBD, which forwards to another, etc. Only after the client's DBD has received an OK message from another DBD does it send an OK message to its client. The DBD does the same type of thing when accumulating ACTD commands: as it queues ACTD commands, it tells another DBD about it, and it dequeues the ACTD request, and receives a response from the ACTD, then it tells the other DBD to remove that particular command from its ACTD "to do" list. If the client detects that its DBD has crashed (because it no longer responds to query requests), it would start a conversation with another DBD. The other DBD would then act as the client's DBD, and would process all of the client's requests.

This whole system ensures that if any DBD were to fail, no data would be lost, and no system changes would be postponed or lost. Each DBD is kept almost completely up to date. If a DBD dies, the client's RDCP *plug* end can connect to another DBD, and continue its conversation as if it had never been interrupted. The client program (the user of the *plug*) neither knows nor needs to know that its DBD connection has changed.

3.5 Database Daemon to Database Interface

The DBD uses a small set of function calls which are built "on top of" Berkeley NDBM [BER86]. These functions enable the DBD to concentrate on its job of servicing client requests, rather than having to know intimate details about the database. The function package allows the DBD to read and write whole user records from/to the database, either in raw text or struct format, whichever is most convenient. Since the DBD's dependence on NDBM is slight, it would be easy to port the DBD to other C-function-oriented database packages. Only the DBD interface to the database would need to be rewritten.

4 Activity Daemon

4.1 Introduction to the ACTD

The Activity Daemon is the part of the Asmodeus system which changes the system database files */etc/passwd*, */etc/group*, creates new accounts, and so on. Each machine served by Asmodeus runs exactly one ACTD, which services requests **only** from DBDs. A DBD and an ACTD will communicate when the DBD has some work for the ACTD to do, as in the section 2 example of jones changing her password. In the example, when the */bin/passwd* client tells the DBD to change jones' password, the DBD changes its local database. It also makes a note for itself (in the list it is keeping about jacob) that it must talk with jacob's ACTD, and send the following command:

```
change passwd Login_name jones Password LKprau6SUwhZo
```

There are two types of commands understood by the ACTD: rough control commands and fine control commands. The rough control commands include such messages as "add user" and "remove user." The fine control commands include some basic file system commands, such as "copy file" and "make directory," as well as commands for changing Asmodeus-controlled files, as in the previous example.

4.2 More on the ACTD

There are (and will be) many programs which use the DBD, but never use the ACTD at all. This is because some of these programs will only **read** data from the DBD, and thus need not be concerned with the ACTD. There also will be programs which change the Asmodeus database(s), but do not affect the ACTD, because the information which they modify in the DBD is not marked as a type of data to export to ACTDs. An example of such a program would be one which tallied connect-time, CPU minutes, and other accounting information, and forwarded it to a DBD. The DBD would store this information, and then another program (perhaps run on a monthly basis) would query the DBD for it and send bills for service.

The programs which affect the ACTD are those programs which need to **change** data in standard UNIX databases, like */etc/passwd* and */etc/group*.

A complete grammar for DBD to ACTD communication is given in Appendix A.

5 Generic Daemon (A Daemon Builder's Kit)

GENERICD is the code for the *acceptor* end of the RDCP communications channel, combined with code for binding to a *socket*, dissociating a daemon process from its controlling terminal, etc. To build a daemon using the GENERICD system, the programmer must:

- Define the character string *server_name*. The DBD defines this to be "dbd."

- Define the *init_daemon* function, which performs whatever initialization the particular daemon needs. The *init_daemon* function must return a unique non-zero integer for each problem situation it can encounter, or zero for success.
- Define the *process_packet* function.

In the Asmodeus system, both DBD and ACTD use the GENERICD interface. The importance of the GENERICD system is that when a person wants to build a daemon which speaks RDCP, much of the work is already done. In essence, the user of GENERICD writes a library of functions, then gives GENERICD enough information to drive it.

6 Reliable Datagram Communications Protocol



Figure 3: The RDCP Protocol

6.1 Definition of RDCP

All parts of the Asmodeus system use RDCP to communicate with one another. There are always two distinct ends of an RDCP communication: a *plug*, or client end, and an *acceptor*, or server end. The *plug* always initiates communication, although either end may decide to terminate the conversation. If the *acceptor* does not reply, the *plug* notifies the client program, which may then decide how to proceed.

The RDCP protocol is **reliable** in that the *plug* will resend packets multiple times, until it gets a response or decides that the *acceptor* is unavailable. The *acceptor* will not forward identical packets to the daemon which it services, and will reply in place of the daemon when it knows the correct response. Thus the communication between the client program and the daemon is one-to-one.

6.2 Implementation of RDCP

The RDCP protocol used in the Asmodeus system is a set of library routines placed “on top of” Berkeley UDP-IP [HP86], an unreliable packet-transfer mechanism. Each packet which is transferred between a *plug* and *acceptor* contains a major packet number, a retry number, and a message. The code which implements the *plug* end of the protocol first *binds* to a privileged *socket*, then resends each packet to the *acceptor* until it either receives

a response, or decides that the *acceptor* is not responding. The *plug* knows that it has received a response to its current query when the packet it receives is marked with the same major packet number that it is currently sending out. The restrictions on RDCP are as follows:

- All conversations start with the *plug* initiating communication using an open conversation packet which has a major packet number of one.
- Each different packet that the *plug* sends has a different, strictly ascending, packet number.
- Each resend of a packet has the same message text.

These restrictions provide the *acceptor* with enough data to determine when a malicious process has inserted packets into the packet stream. Although the *acceptor* can determine that a malicious process has inserted an unauthorized command into the packet stream, it is impossible for it to tell which command is the offender, meaning that the unauthorized packet may have already been processed by the daemon. This system is not any more of a security hole than the presence of terminal concentrators on a network, because this system detects and logs discrepancies. We hope to make this sort of deception of a client or server very difficult by using a public-key encrypted signature on each packet transferred.

6.3 Asmodeus System Dependency on RDCP

The Asmodeus system does not depend specifically on the RDCP protocol, but on a reliable transfer mechanism which must look like a packet interface. It also depends on the fact that the communications channel between the client program and the daemon is reliable. It would be possible to implement the client and daemon views of the RDCP protocol using TCP-IP, if desired, with some drawbacks. We had several reasons to believe that TCP-IP was not appropriate for our application, since we wanted the following features, which were not available with it:

- Ability to trap the knowledge of the death of a “connected” server by the Asmodeus function package, without the client program having to know,
- More customizable retry control,
- A real packet-level interface, without having to resort to artificial boundaries in a stream,
- Ability for a single DBD or ACTD server to serve many hundreds or thousands of clients simultaneously.

7 Differences Between Asmodeus and YP

At first glance, Asmodeus seems to be a direct replacement for *YP*, but this is not the case. The services which Asmodeus provides do not overlap completely with those of *YP*. Asmodeus works behind the scenes, so that only programs which **modify** Asmodeus-controlled files are affected **at all** by Asmodeus' presence. This is possible because a local copy of each of the Asmodeus-controlled files exists on each Asmodeus-controlled system. The basic differences between Asmodeus and *YP* are summarized below:

- *YP* needs Unix kernel modifications for efficient operation. Asmodeus runs completely in user space, so that it can be used with a Unix offered only as a binary distribution, although currently all Asmodeus programs must have "root" privileges.
- *YP* understands multiple domains [SUN86b], and Asmodeus does not.
- *YP* remaps many system calls into network calls, which is often very slow. Asmodeus remaps no system calls.
- *YP* provides many disk and file services, where Asmodeus provides only a very small number of remote file functions (see Appendix B).
- Since *YP* is implemented in part as a new set of *libc* library calls, whenever a C program uses any standard I/O routines, it includes a good part of *YP*, which makes the size of executable code grow dramatically. In Asmodeus, new function calls are only included by programs (like */etc/passwd*) which need them.
- Asmodeus has an intimate understanding both of the data it is handling, and of the relationships between the data. This enables it to be efficient. *YP* has **no understanding** of the data it handles [SUN86c].
- Programs which just **read** Asmodeus-controlled files need not be changed at all, whereas programs which read *YP*-controlled files still must carry the weight and speed penalties of *YP*-modified *libc* function calls, as well as the speed penalties of network-modified system calls.
- Programs which **modify** Asmodeus-controlled files (password, group, mail-aliases, etc.) must be rewritten or discarded to be used with Asmodeus; with *YP* almost nothing need be rewritten, just recompiled.
- If the Asmodeus system fails, it only affects the average user *jones* when she wants to change Asmodeus-controlled files (using *chfn*, *chsh*, or *passwd*). If the *YP* system fails, the effects are much more noticeable (i.e. the "ls" command can fail if *YP* fails).

If we take the case of a change in a password file as a typical case of *YP*-Asmodeus functional overlap, it is evident that Asmodeus does things differently than *YP*. When the *YP* system is in use, it detects that the file has changed, and ships the **entire file** across the network [SUN86b] to other machines (using *ypxfer*). When the Asmodeus system is in use, a program like */etc/passwd* has to talk with the DBD to change a password file; the DBD then

ships a much smaller amount of information across the network to one or more ACTD(s) to effect the change.

YP is an elegant, general solution to the problem of a widely available replicated database, whereas Asmodeus is a specific solution, tailored for systems administration. Asmodeus was designed to always be correct (completely up to date) for queries on its database. In a steady-state situation, *YP* and Asmodeus are functionally equivalent in the areas where they overlap; it is when we discuss database updates that the differences in goals and strategies of the two designs become evident.

Asmodeus is more transparent to the average user *jones*, because she does not see Asmodeus unless it fails while she is using one of the few programs which modify standard system databases. Since it does not remap system calls, no standard Unix programs (like “ls”) are affected by its presence; due to its understanding of the data content, it can afford to send fewer and smaller packets across the network.

Appendix A – Grammars for Asmodeus Communication

The communication between Asmodeus clients and the DBD, and between the DBD and the ACTD, is transferred over a reliable packet-level medium. The complete grammar of these packets is given below. The words in “<>” can be thought of as variables in the grammar, whereas the words given without brackets are constants. An example will perhaps better clarify the distinction.

Given a grammar line

```
find Login_name <existing_user_name>
```

and the following strings,

```
find Login_name jones
find Login_name "root"
find Login_name "Barney Rubble"
LaTeX is useful
```

any of the first three lines would fit the grammar, although the third might not work in practice—it has been empirically determined that “Barney Rubble” is not a valid login name at our site. The fourth would not, because the constant terms in the line do not match the constant terms in the grammar.

The point here is that there is an understanding between each half of an Asmodeus communication as to what is syntactically valid, and this understanding is encapsulated in the following grammars.

Client to DBD Communication

```
add Login_name <existing_user_name> Machine_name <single_token>
```

```

add Login_name <single_token>
change Login_name <existing_user_name> <root_field> <single_token>
change Login_name <existing_user_name> <single_token>
change Login_name <existing_user_name> Machine_name
    <existing_machine_name> <machine_field> <single_token>
change Login_name <existing_user_name> Machine_name
    <existing_machine_name> <single_token>
find Login_name <existing_user_name>
find Login_name <existing_user_name> Machine_name
    <existing_machine_name>
first Login_name
first Login_name <existing_user_name> Machine_name
lock <existing_user_name>
next Login_name <existing_user_name>
next Login_name <existing_user_name> Machine_name
    <existing_machine_name>
read Login_name <existing_user_name> <root_field>
read Login_name <existing_user_name> Machine_name
    <existing_machine_name> <machine_field>
read_raw <existing_user_name>
unlock <existing_user_name>
write_raw <existing_user_name> <single_token>

```

DBD to ACTD Communication

```

add alias <new_alias>
add group <new_group> <new_group_ID>
add passwd entry <new_user_name> <single_token> <new_user_ID>
    <existing_group_ID> <single_token> <new_file> <existing_file>
add to alias <existing_alias> username <existing_user_name>
add to group <existing_group> username <existing_user_name>
add username <new_user_name> <single_token> <new_user_ID>
    <existing_group_ID> <single_token> <new_file> <existing_file>
change filename <existing_file> group <existing_group>
change filename <existing_file> owner <existing_user_name>
change filename <existing_file> to <new_file>
change group ID <existing_group> <new_group_ID>
change group name <existing_group> <new_group>
change passwd <existing_user_name> <password_field> <single_token>
change username <existing_user_name> <new_user_name>
close
copy filename <existing_file> <new_file>
open
remove alias <existing_alias>

```

```

remove filename <existing_file>
remove from alias <existing_alias> username <existing_user_name>
remove from group <existing_group> username <existing_user_name>
remove group <existing_group>
remove username <existing_user_name>

```

Appendix B – Activity Daemon Functions

ACTD Fine controls

addToPasswd (entry) – Adds a complete entry into the password file, assuming that one under the same login name and/or user ID, does not already exist.

editPasswd (login, field, newData) – Changes information in */etc/passwd*. It searches for the entry “login” and replaces the specified field with “newData.” This is used by the ACTD in response to such DBD clients as *chfn*, *passwd*, and *chsh*.

changeOwner (login, file) – Exactly like *chown*.

changeOwnerDir (login, directory) – Recursively descends a directory, changing ownership of files and directories.

changeGroup (GID, file) – Exactly like *chgrp*.

changeGroupDir (GID, directory) – Recursively descends a directory, changing the group of the files and directories.

addToGroup (login, group) – Adds a user to an existing group.

removeFromGroup (login, group) – Removes a user from an existing group in */etc/group*.

makeGroup (groupName, GID) – Adds a group to */etc/group*.

removeGroup (groupName) – Removes a group from */etc/group*.

changeGroupName (oldName, newName) – Changes the name of a group in */etc/group*.

changeGID (groupName, newGID) – Changes the GID of “groupName” to newGID.

moveDir (oldDir, newDir) – Changes a directory name.

deleteDir (directory) – Removes a directory.

makeDir (directory) – Creates a directory.

moveFile (oldName, newName) – Changes a file’s name.

copyFile (sourceFile, destFile) – Copies “sourceFile” to “destFile.”

deleteFile (filename) – Removes the file “filename.”

makeForward (login, machine) – Creates a *.forward* file for the user “login” that points to “machine.”

createFile (filename) – Creates a zero-length file.

addToAlias (login, alias) – Adds the user “login” to the “alias” mail-alias.

post() – Complete the update(s) in progress. Used at end of session.

ACTD Rough Controls

addUser (fullName, login, userID, groupName, shell) – Adds a new user to the machine. It does a `getgrname(groupName)` to find the proper GID. Then it does a `getpwnam(login)` to make sure that the login is not in use. It assumes that the UID given to it by the DED is unique. It then uses other ACTD functions to create the entry in the `passwd` file, by calling `addEntry()`, then `makeDir()`, then `copyInits()`, and finally `changeOwnerDir()`. It also keeps a log of its actions.

copyInits (homeDir) – Copies all of the “dot files” that a user needs, such as *.login*, *.logout*, etc. It uses `copy()` to do its work.

deleteUser (login) – Removes a user from the machine. Searches */etc/passwd* for “login,” removes that entry, then does an `rm -r` on the user’s home directory and mail spool file.

changeUserGID (login, newGID) – Changes the GID of a user, and changes the appropriate files.

changeUserID (login, newUID) – Changes the UID of the user, and changes the appropriate files.

Appendix C – Internal Structure of the Asmodeus Database

Asmodeus stores a “root field” for each user which it knows about. This field stores all of the information which we are certain will not change for a user, whichever machine he is using. There is also one record in the database for each account that a user has on a particular machine. The formats of both of these record types follow:

Key field
login

Content field

```
login:userID:groupID:fullName:studentID:mailMachine:
homeAddress:homePhone:workAddress:workPhone:departmentSupport:
supportLimit:balanceDue:balanceDue1Month:balanceDue2Months:
balanceDue3+Months:machine_1, machine_2, ..., machine_N:
```

Key field`login@machine`**Content field**

```
machine:encryptedPassword:lastLogin:activationDate:
expirationDate:accountActive:accumulatedCPU:totalDiskSpace:
oldDiskSpace:accumulatedConnect:printerUse:
homeDirectory:shellName:
```

References

- [BER86] "UNIX Programmer's Reference Manual (PRM), 4.3 Berkeley Software Distribution Virtual VAX-11 Version," Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- [HP86] "ARPA Services/300 User's Guide," Hewlett-Packard Company, 1986, pp. 4-1 - 4-79.
- [HP87] "HP-UX Systems Administrator's Manual, HP 9000 Series 300 Computers," Hewlett-Packard Company, 1987, pp. 227-278, pp. 306-380.
- [LEF86] Leffler, Samuel J., Fabry, Robert S., Joy, William N., Lapsley, Phil, Miller, Steve, Torek, Chris, "An Advanced 4.3BSD Interprocess Communication Tutorial," in *UNIX Programmer's Manual, Supplementary Documents 1 (PS1), 4.3 Berkeley Software Distribution*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- [SEC86] Sechrest, Stuart, "An Introductory 4.3BSD Interprocess Communication Tutorial," in *UNIX Programmer's Manual, Supplementary Documents 1 (PS1), 4.3 Berkeley Software Distribution*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- [SUN86a] "Commands Reference Manual, Sun Release 3.2" Sun Microsystems, 1986, pp. 516-519.
- [SUN86b] "Systems Administration for the Sun Workstation," Sun Microsystems, 1986, pp. 35-60.
- [SUN86c] "Yellow Pages Protocol Specification," in *Networking on the Sun Workstation*, Sun Microsystems, 1986.
- [TEK87] "UTek System Administration," Tektronix Inc., 1987, pp. 5-1 - 5-13.

Big Brother: A Network Services Expert

*Don Peacock and Mark Giuffrida
markg or donp @caen.engin.umich.edu*

*Computer Aided Engineering Network
University of Michigan
229 Chrysler Center
Ann Arbor, MI 48109-2092*

ABSTRACT

At the University of Michigan, the monitoring of network services in a large heterogeneous network is becoming an automated process with the advent of "Big Brother." The system can identify a network failure and take a specified set of actions to correct it.

1. Introduction

Network services are becoming too complex for individuals to comprehend completely. This is largely due to the very high number of interdependencies between the elements in a large network. For example, to remotely login to our Alliant FX-8 from one of our Apollos, the following resources must be operational:

- Apollo host TCP server
- Apollo token ring
- Apollo Ethernet gateway
- Ethernet Proteon fiber gateway
- remote Ethernet gateway
- Alliant inetd daemon

If any one of these resources is unavailable, then the remote login service is unavailable. On a typical network, there are many critical services: mail, NFS, route and gateway daemons, name servers, etc. The knowledge for these services is typically shared between a large systems staff. When a service fails, a knowledgeable staff member must be located to fix the problem.

Networks need to be able to detect and perform their own repairs. If a network facility could pinpoint the specific resource causing the problem, it could notify a staff member with the appropriate information or fix the problem itself. The key elements in allowing a network to fix its own problems are: (1) a central knowledge and rules base, the *configuration file*, and (2) a network server to utilize the configuration file, *Big Brother*. Most Unix systems have tools that allow monitoring of any system service and other tools to repair those services. By encapsulating these tools into a rules base, Big Brother can detect and correct failures. This integration allows the network to monitor its own services, thus guaranteeing a more reliable and consistent framework of network services.

2. History and Current Status

The original motivation for designing Big Brother was to write a server that would monitor network services and notify staff members when problems arose. The requirements rapidly grew into the following:

- The ability to keep state information on monitored resources.
- The ability to use any existing program as a check for determining a resource's state.
- The ability to provide a specified set of actions (as opposed to just notifying) when a problem arises. Examples of this are recording the incident in a log, changing a status monitor, sending electronic mail, attempting to correct the problem, etc.
- The ability to use dependencies when checking the state of a resource. For example, if a Sun is being monitored across a gateway, then the gateway itself must be up in order to check the state of the Sun.
- The ability to be maintained by all systems staff.

Big Brother currently implements all of the above requirements. It runs on BSD-based Apollos, Suns, Vaxen, and Alliant computers. It is written in C and lex. The lex-generated portion is used in both the expression evaluation and interpretation of the configuration file.

Big Brother is in use at the University of Michigan on an Apollo DN4000 workstation. It monitors the main Apollo ring of 350 nodes, two bridged Apollo rings of 50 nodes, the local building Ethernet, two remote Ethernets, an SMTP mailer on the Apollos, an Alliant FX-8, and a Sun fileserver. When a problem is detected with one of these monitored resources, it will log the incident and execute a program, which connects with a modem to page an appropriate staff member. The pagers we use display codes, which represent a particular resource. For example, the code "111" is sent when a failure is detected with the Apollo token ring and the code "000111" is sent when the failure has been corrected. All other resources we monitor have their own specific code and set of people to notify when a failure occurs. Thus, the responsibility for monitored resources is distributed amongst the knowledgeable staff.

Big Brother is also used to monitor current availability of workstations in the student labs. A user can call up a Dectalk™ unit interfaced to another server and be told over the phone where open workstations are located. This is accomplished by having Big Brother log information about its monitored resources into a set of files, which the Dectalk server uses. Big Brother is also used to monitor the disk space of all major file servers and it will send electronic mail to specified staff when disk space reaches a threshold level.

3. Internal Implementation

Big Brother is normally set up to watch many system services. Each service is usually made up of numerous resources. Each resource is an object in a Big Brother configuration file. For example, a service such as SMTP relies on several resources. Some of these resources might be the SMTP daemon, the TCP server, and gateway connections. During execution, Big Brother will process each of the objects sequentially. When the last object is processed, it sleeps for a default twenty seconds before continuing with the next pass.

A Big Brother configuration file is best understood through an example. The following is a sample definition for monitoring an Alliant FX-8. The host Big Brother is running on is an Apollo workstation whose token ring is gatewayed to the Ethernet where the Alliant resides.


```

/* Alliant FX-8 */
object = alliant
  testcmd : /usr/local/bin/ping -telnet rocket
  state   : [ {alliant.testnum} > 8 ]
  if [ !(alliant.state) ] then
    execute /usr/local/bin/page 5551212 222
  endif

```

The first statement in this resource definition is the *object* statement. In this example, the object has been appropriately named *alliant*. When an object is evaluated by Big Brother, the *testcmd* statement is executed first. The return code and the number written to standard output from *testcmd* are saved and can be retrieved by specifying an appropriate variable. Variables can appear anywhere and are enclosed in braces. Within these braces are two words separated by a period. The word to the left of the period is the object name and the word to the right is a special keyword. The keyword *testnum* is used to retrieve the number written to standard output from the *testcmd*. Thus, in this example, *{alliant.testnum}* would be specified to retrieve the *testcmd* value.

After the *testcmd* is executed, the *state* expression is evaluated. This will evaluate to true or false and represent the boolean state of the object. The state of an object can be referenced later using the variable keyword *state*. In this example, the state expression tests to see if the number written to standard output by the ping command is greater than eight. The local ping program will write a number greater than eight to standard output if it can reach the host. Thus, the state of this object is set to true if the Alliant responded to the ping.

Once the state expression is evaluated the *if*, *log*, *execute*, and *set* statements associated with the object are executed in order. The *if* statement in this example tests to see if the state of the *alliant* is false (i.e., if it cannot be pinged). If so, the enclosed *execute* statement is performed. An *execute* statement will cause a *vfork* and *execl* of the specified command. In this example, a pager program is invoked and it will connect to a modem and page a staff member with the Alliant's failure code of "222".

There are several problems with this object definition. Since Big Brother is running on an Apollo connected to a token ring, it is dependent on that token ring. If the token ring fails a staff member will incorrectly receive a page that the Alliant is down. The gateway connecting the token ring to the Ethernet is also a dependency. The desired behavior would be to page when the ping fails, but only when the token ring and gateway are up. This dependency problem is corrected by using a *depends* expression. Big Brother evaluates the *depends* expression of an object before running the *testcmd*. If the expression evaluates to false, the rest of the processing for that object stops and Big Brother goes on to the next object. For this example, assume that there are two more objects being monitored. The first of which is the token ring, *apolloring*, and the second is the gateway node, *gateway*. Also assume that the state kept for these objects are set true when they are functional. The successful pinging of the Alliant would depend on the variables *apolloring.state* and *gateway.state* being true. The correct *depends* expression for the Alliant, therefore, would be:

```

depends : [ (apolloring.state) && (gateway.state) ]

```

Another problem with this object definition is that Big Brother will page each time it evaluates the Alliant to be down. User defined variables are used to correct this problem. They have the same syntax as system variables except that their names cannot contain periods. User defined variables are given a default value of false and can be changed using the *set* statement. A more complete definition of the Alliant is as follows:

```

/* Alliant FX-8 */
object = alliant
depends : [ { apolloring.state } && { gateway.state } ]
testcmd : /usr/local/bin/ping -telnet rocket
state : [ { alliant.testnum } > 8 ]
/* Is the alliant down? */
if [ !{alliant.state} ] then
    /* Has a page already been issued? */
    if [ !{alliantpage} ] then
        log { .date } Alliant is down, paging code -> 222
        execute /usr/local/bin/page 5551212 222
        set alliantpage = [ .true. ]
    endif
endif
/* Is the alliant up and a page been sent out for it being down? */
if [ {alliant.state} && {alliantpage} ] then
    log { .date } Alliant is back up, paging code -> 000222
    execute /usr/local/bin/page 5551212 000222
    set alliantpage = [ .false. ]
endif
endif

```

The above example also includes a *log* statement, which will record the incident. The date and time of the incident are also recorded by the use of the predefined internal variable *{.date}*.

Big Brother was designed to be robust under the worst conditions. A built in safeguard allows *testcmd* a default sixty seconds to execute before it is killed. *Execute* statements cause the specified programs to detach and run in the background. Thus, no tool used to monitor and correct network services will hang Big Brother itself.

4. Future Work

Currently, there are several directions we are investigating for future enhancements. The first is to allow communication between cooperating Big Brothers. This would allow a Big Brother on one machine to reference objects being watched by a Big Brother on another machine. It would also eliminate the single point of failure of running one Big Brother. This communication can be accomplished in a number of ways. One method is to provide a socket-based query mechanism, which allows information to be selectively extracted from the monitor. A second method would be to integrate Big Brother into a distributed database. This database would be responsible for containing all state information associated with all Big Brothers.

Another useful feature would be to communicate scheduled changes in network services such as downtime to Big Brother. Both socket-based communication and database queries are being considered for this enhancement as well.

Another direction we are looking into is a Big Brother that would support multiple threads of control. The resources monitored would be arranged in a tree, which would reflect their dependencies. Each thread of control would operate on its own branch of the tree. When a dependency is broken, all the subordinate threads of control would be suspended. Conversely, when a broken dependency is restored, the subordinate threads of control would be restarted.

5. Why Not Shell Scripts?

We feel that Big Brother offers many advantages over conventional shell scripts. One such advantage is performance. Big Brother is only partially interpretive. Its configuration file is read into internal format at startup and is reread automatically upon modification.

Big Brother's syntax can easily be changed by replacing the appropriate lex module. This allows for the possible development of a standardized syntax for monitoring objects.

Shell scripts are usually much larger than a corresponding Big Brother configuration file. Maintaining timeouts and keeping object statistics would make shell scripts large and repetitious, or very cumbersome.

The future enhancements we envision would be non-trivial, if not impossible, using shell scripts.

6. Conclusion

Network services are becoming too complex for individuals to completely comprehend. Monitoring of services is better accomplished by the network itself. Big Brother combines expertise from many individuals to use in monitoring network services and correcting failures.

7. Appendix: Big Brother Configuration File Syntax Summary

The only special characters in the following syntax are "<>", which denote a substitution. Boldfaced characters represent literal strings.

```
object    = <objectname>
depends    : [<expression>]
testcmd   : <cmd>
state     : [<expression>]
```

```
if [<expression>] then
    <stm>
endif
if [<expression>] then
    <stm>
else
    <stm>
endif
```

```
log        <text>
execute    <cmd>
set        <uservar> = [<expression>]
/*<comment>*/
```

<objectname>	any ASCII string (periods and white space not allowed)
<cmd>	any command and its arguments (variables allowed, see <var>)
<text>	any ASCII text (variables allowed)
<stm>	one of log, execute, set, if statement
<comment>	any ASCII text (nested comments not allowed)

<expression>	<i>C-like arithmetic syntax expression (variables allowed)</i>
<var>	<i>one of <objectvar>, <uservar>, <internalvar></i>
<objectvar>	<i>{<objectvar>.<field>}</i>
<uservar>	<i>any ASCII string (periods and white space not allowed) enclosed in {}</i>
<internalvar>	<i>{.<intfield>}</i>
<field>	<i>one of state, testnum, testrc, min, passes</i>
<intfield>	<i>one of log, config, start, passes, time, sec, min, hour, day, month, year, wday, yday, isdst</i>

8. Acknowledgements

Thanks are due to Mike Hucka, Randy Frank, Paul Anderson, John Muckler, Eric Wolf and other supporting members of the systems staff for their help.

